(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A MODEL OF SOFTWARE MAINTENANCE
FOR
LARGE SCALE MILITARY SYSTEMS

by

Isaak Mostov

June, 1990

Thesis Advisor: Luqi

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 37	7a. NAME OF MONITORING ORGANIZATION: Computer Science Dept. Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A MODEL OF SOFTWARE MAINTENANCE FOR LARGE SCALE MILITARY SYSTEMS

12. PERSONAL AUTHOR(S)
Mostov Isaak

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master Thesis	FROM ____ TO ____	1990 June	86

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Deaprtment of Defence or the U.S. Government

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Software maintenance, Managing software maintenance, Maintenance-oriented engineering database, Modelling of maintenance process.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The maintenance of large military software systems is complex, involves users as well as software professionals, and requires appropriate management, which is one of the most important factors in efficient maintenance. Maintenance management requires information about the current state of the maintenance process that should be organized within a maintenance-oriented Engineering Database. This database should include all the necessary data about software changes, system configuration, maintenance task scheduling, etc., and it should be based on a realistic model of the maintenance process.

This thesis proposes a mathematical Model of Software Maintenance that uses graphs to model the relationships between maintenance tasks and software components. The Model addresses the dynamic behavior of the maintenance process and supports priority and precedence of maintenance activities.

The proposed Model of Software Maintenance provides a sound basis for implementation of a maintenance-oriented engineering database that supports automation of maintenance management, e.g., process control, task scheduling, job assignments, planning and forecast, gathering and interpretation ofmaintenance statistics and metrics, etc.

(KR) ←

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Luqi	22b. TELEPHONE (Include Area Code): (408) 646-2735; 22c. OFFICE SYMBOL: CS/Lq

Approved for public release; distribution is unlimited.

A Model of Software Maintenance
for
Large Scale Military Systems

by

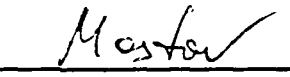
Isaak Mostov
Major, Israeli Air Force
B.S.C.S., The Hebrew University of Jerusalem

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

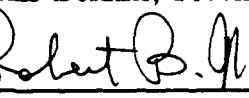
Author: 

Isaak Mostov

Approved by: 

Luqi, Thesis Advisor

Valdis Berzins, Second Reader



Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

The maintenance of large military software systems is complex, involves users as well as software professionals, and requires appropriate management, which is one of the most important factors in efficient maintenance. Maintenance management requires information about the current state of the maintenance process that should be organized within a maintenance-oriented Engineering Database. This database should include all the necessary data about software changes, system configuration, maintenance task scheduling, etc., and it should be based on a realistic model of the maintenance process.

This thesis proposes a mathematical Model of Software Maintenance that uses graphs to model the relationships between maintenance tasks and software components. The Model addresses the dynamic behavior of the maintenance process and supports priority and precedence of maintenance activities.

The proposed Model of Software Maintenance provides a sound basis for implementation of a maintenance-oriented engineering database that supports automation of maintenance management, e.g., process control, task scheduling, job assignments, planning and forecast, gathering and interpretation of maintenance statistics and metrics, etc.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I owe a considerable amount of gratitude to my advisor, Professor Luqi. Her guidance and encouragement were instrumental in the completion of this research. She was the one that made me start this research in the early stages of my studies at the Naval Postgraduate School and followed with timely advice and support during the long periods of conception and development of the Model of Software Maintenance.

I also am grateful to my second reader, Professor Valdis Berzins, for his willingness to evaluate and discuss raw ideas during the process of crystallization of the more sophisticated parts of the Model of Software Maintenance.

A special acknowledgment is due to Professor Kim Hefner from the Department of Mathematics. I owe her a great deal for teaching me the basics of graph theory and for her unique help in defining the mundane mathematical details of the Model in correct form and substance.

In addition, I would like to thank Professor N.F. Schneidwind, Professor T. Abdel-Hamid, Professor Moshe Zviran, Dr. Kraemer, Professor Man-Tak Shing, Lt.Cmdr. J. Yurchak, USN and Professor Dan Beery for their help in revising parts of my thesis and for their constructive suggestions.

There is no doubt in my mind that without the help and the cooperation of all of these people, who contributed their precious time to review my work, it would be impossible to develop the Model of Software Maintenance to its present state.

Finally, but by no means any less important, I would like to acknowledge the unique contribution of my wife, Judith. Without her uncompromising support and encouragement it would be very hard for me to devote the necessary time and energy to complete this research on time.

I. SOFTWARE MAINTENANCE IN MILITARY SYSTEMS

Software maintenance is the longest and costliest phase in the lifecycle of a large scale computer systems, especially in military use. The United States Department of Defense invested in software about 5 billion dollars in 1985 (estimated to reach 20 billion dollars in 1990s), approximately 60-70% of which were spent after it had been tested and delivered to users, i.e., in software maintenance [Ref. 1]. Due to the amount of resources involved, it is clear that any improvement in software maintenance is important and beneficial, especially for military organizations that are dependent on computer systems for their daily operations.

This chapter presents the distinguishing characteristics of military software systems, main issues that arise in their maintenance and outlines requirements for maintenance-oriented engineering database aimed to improve the process of software maintenance.

Although in general Software Engineering problems in military computer systems are the same as in commercial systems, the characteristics of the computer systems used by military organizations complicate the software maintenance tasks considerably. Understanding the specific characteristics of large military computing systems helps in understanding the complexity and addressing the problems of the software maintenance in general.

A. SOFTWARE SYSTEMS IN MILITARY USE

Software systems in military use can be grouped into the following major categories:

- Embedded Software Systems.
- Command, Control and Communication (C^3) Support Systems.
- Data Management Systems.
- Local Information Systems.

An embedded software system is one in which software is only a part (and usually not the major one) of the system. An example for such a system is a guided missile or an aircraft flight control subsystem. The embedded software system is characterized by the following:

- The capabilities and performance of the system as a whole are determined by various technologies and not primarily by their software component.
- The system is influenced directly by changes in the battlefield environment (i.e., introduction of new adversary weaponry, ECM Systems, etc.) and rapid adaptation to the new environment is essential.
- Although the system's usefulness and survivability as a whole is influenced by the operational environment changes, usually it is the software component that is the main factor in adaptation of the System to the new environment. The main reason for this is due to the fact that system's intelligence is usually concentrated in its software component, and therefore it is (relatively) easy to change and install the updated version in the operational system/equipment.
- In order to improve performance and to make the most efficient use of existing resources (which are always limited due to size/weight considerations of the whole system), special low level hardware features and time/space optimization are utilized.

A Command, Control and Communication (C^3) support system is one in which a computer system performs a defined operational task that serves an entire organization. These systems have the following characteristics:

- The system's architecture and software are customized and are organization-dependant.
- The system becomes a part in the daily function of the organization and performs (or helps to perform) operational tasks and duties of the organization.
- Failure of the system has an immediate influence upon the organization's operational capabilities.
- The system has some real time properties and its software is (usually) the single most important factor in the system's capabilities and performance.
- Due to the characteristics of C^3 support systems mentioned above, these systems are very large, extremely complicated, and require very large resources and effort in development and maintenance.

The data management systems category includes the traditional database systems which manage the organization's information. Such systems, however sophisticated and complicated, do not have a direct and immediate influence upon operational capabilities of the military

organization, i.e., they do not have an immediate effect in the "battlefield situation" that the organization may find itself in. However, they are very important to their organizations and usually have a very long life span.

The local information systems category includes systems that are targeted for local use and that have limited (but possibly important) influence on the organization as a whole. Personal computer based systems with small-scale stand-alone applications serve as a good example for a computer systems in that category. The local information system category can be characterized by the following:

- The use of the system is restricted to specific tasks for benefit of some department of the organization.
- The software in the local information system is relatively small and simple and it is built by using mostly "off the shelf" products and components.

The first three categories are the most important for any military organization (that has them) and they are characterized by their staggering costs and longevity. Also, unlike some commercial applications, the military computer systems in these categories are characterized by the speed they must adapt to a changing environment, therefore giving a very high priority to maintenance tasks that incorporate the necessary changes and upgrade the system.

Local information systems do not differ from commercial computer systems, have virtually no special influence upon problems of software maintenance in the military computer system, and are not in the scope of this work.

B. MAINTENANCE OF MILITARY SOFTWARE

Due to its characteristics, the military computer system and its software has a somewhat complicated life cycle. This life cycle can be divided into two major parts: initial development and the following maintenance.

The development cycles of software and hardware components of the systems are very similar (at least for specification, requirement and system design phases). After all, there is not

much difference in the methodology of developing "from scratch" electronic components and of "bug proof" software. In both cases, initial requirements and constraints are translated into functional specifications. An architecture of the system is designed to implement the functional specifications and then the design is built and tested against the initial requirements and constraints. If any of the requirements and the constraints are not met, the development team goes back to the "drawing board" and improves or changes its initial design. After the system has been completed and accepted by the user organization, the development teams are dispersed or diverted to other systems. Sometimes the team begins to work on an improved version of the delivered system.

The maintenance phase of the system's life cycle introduces the major differences between the system's software and hardware components.

The electronics and the mechanics of the system are maintained according to various organizational policies in order to insure the designed capabilities and performance. Usually, no engineering changes are done to the system unless a disastrous design problem has been discovered after system delivery, and, specifically, the system design is usually not changed by any maintenance activity.

This is not the same with the software. Since software is the most flexible part of a system, it is usually the case that software is changed to accommodate alteration in the operation environment and to improve the system's performance and reliability. This process is called ***software maintenance***. Software maintenance is defined as "performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production[1]" [Ref. 2], and it is intended to correct undiscovered faults, to improve performance or other attributes, or to adapt the product to a changed environment [Ref. 3]. Due to the fact that the maintenance phase begins after the software system is delivered to the user

---

[1] The meaning of "placing the system into production" here corresponds to "making the system operational".

organization (i.e., when it actually becomes property of user organization), the user becomes responsible for the system capabilities and performance and can change it at will.

The need for changes in military computer system arises from operational experiences that are accumulated while operating the system for some period of time. These changes also include specifications and performance discrepancies and previously undiscovered bugs. Usually, (especially in the case of the C^3 Support Systems) software changes are induced by the system itself due to the impact it has upon an organization's operational philosophy, methods, and procedures. The trend of these changes is to integrate new and compatible functions to the system.

Thus, besides the regular "bug fixing", software maintenance consists of engineering changes to the original design of the system and is actually a redesign of the system [Ref. 4]. These engineering changes to the original design are complex, unpredictable and cannot be accommodated without some iterative process [Ref. 3].

Although the problems in software maintenance are relevant to military and non military software systems alike, software maintenance of military systems is complicated by the following:

- Due to high performance demands and packaging problems, military computer software systems are built placing more emphasis on efficiency than on future maintainability. This phenomenon holds true even with the latest advances in software engineering practices and tools.
- Since military software systems are integrated with hardware to a very large extent and have direct interfaces with specific devices (especially in embedded systems), some modules (or even entire systems) are implemented using low level languages.
- Due to stringent timing constraints found in most embedded and C^3 systems, any change of software due to required maintenance may compromise the real time performance of the system as a whole.
- The field testing of weaponry systems and platforms is expensive and may result in physical destruction of the system. This fact strengthens the need for the best possible testing of the embedded software prior to new release. It also imposes problems in bug detection and repair, especially for fatal bugs in critical software that "surface" in situations that are difficult to predict and simulate.

- The average military software system is very large and complicated (e.g., C^3 Support Systems). In these systems every engineering change to one of the modules may have a "ripple" effect upon other parts of the system.

- Sometimes, due to external factors, a number of different versions of the same system are in operational use by the military organization (e.g., old avionics systems with limited memory capacity are used along with newer systems with upgraded memory and software capabilities). This imposes maintenance on a number of software versions that may be incompatible but are interrelated in their components.

It should be noted that the above characteristics are not unique for the military computing systems. Some commercial software systems may also possess such characteristics, although they are very few in the abundance of existing commercial computing systems and software.

The maintenance problems are further complicated by the following characteristics of the software maintenance team (see [Ref. 5], [Ref. 6] and [Ref. 7]):

- The maintenance team is small (compared to the development team that built the system), e.g., some large scale systems that were developed involving hundreds of man-years are maintained by teams with (usually) less then 10 programmers. Combined with the high volume of the maintenance tasks (especially in the large and complex software systems), this leads to severe limitations on the number of available maintenance programmers and causes a backlog in maintenance activities.

- Few (if any at all) maintenance team members have been involved in any way in the original development phase of the system, thus resulting in low relative development experience. This may result in deficiencies in the team's knowledge about the system, especially on the engineering and architectural levels.

As a result, the task of the software maintenance team is hard and time consuming, and therefore costly. Programmer turnaround as well as trend of maintenance to raise the software complexity level of the military system while increasing its longevity contribute to the high cost of software maintenance [Ref. 8].

Sometimes this high cost is responsible for the decision to abandon the current software system and to develop a new one. This new software, although based upon the original requirements and constraints, is supposed to comply with new requirements and constraints that were deduced from lessons learned while using and maintaining the original system.

C. SOFTWARE MAINTENANCE ACTIVITIES AND PROCESS

Software maintenance activities can be grouped into two major categories [Ref. 9]:

- Software Update.
- Software Repair.

The main difference between these two categories is in the way they reflect upon the original functional specifications of the system: Software Repair does not change them while Software Update does.

Software update activity is actually a response to the addition of new requirements or to the changes of an original system requirement. In military computer systems, it is as important as software repair and it is usually the single most important factor for the system's long life span because, as a result of software update activity, the system is being adapted to the changing operational environment and it continues to provide required services to the military organization it serves.

It involves heavy interaction with the user (or his representatives) and actually goes through the classic software development cycle - from requirements through functional specification and architectural design to implementation and testing. But, unlike the original development cycle, here the software maintenance team does not have "artistic freedom" and must take into consideration the existing system's requirements, constraints, capabilities, and performance. This makes the software update activity the most difficult, demanding and costly of software maintenance tasks. Surveys of maintenance history of complex software systems in military use (e.g., [Ref. 6]) had shown conclusively that the majority of maintenance resources were spent to add new capabilities to system, while the latent defect corrections consumed considerably smaller maintenance resources.

The software repair activities can be further classified into the following categories:

- Corrective maintenance - fixing implementation and design bugs.
- Adaptive maintenance - adapting the system to changes in processing or data environment.
- Perfective maintenance - improving the performance and maintainability of the system.

Since there is no software system without at least one bug, the need for corrective maintenance is easily understood and accepted.

Perfective and adaptive maintenance become necessary in systems where foreseeing all changes (in processing and data environments) and forecasting system performance were based upon guesswork. This is usually the case in systems based upon new technologies with a high level of uncertainty. While building such systems one is concerned with the question "will it work?" rather than "how fast and good will it be?".

The maintenance process life cycle can be divided into various distinct phases [Ref. 2] as it is shown in Figure 1-1.

In the first four phases of the maintenance process, the need for a system's change is formulated and classified. For corrective maintenance problems this is a "touchy" phase because most of the problems are detected by system users as behavior inconsistencies while the system is in operational use. The problems that arise may be an "operator mistake" which should be filtered before the supposed inconsistency is brought to the attention of the maintenance team [Ref. 10]. In order to deal strictly with real problems, the final problem identification and classification should be done by someone with knowledge about system requirements and its expected behavior. Usually, this job requires an experienced maintenance programmer.

For other software activities, i.e., adaptive and perfective maintenance, a formulation of the required system's change is actually a statement of an additional system requirements.

The requirements analysis phase includes problem research and cost estimation. In this phase the "bug" may be "hunted down" or its "natural habitat" identified. For non-corrective

1. *DETERMINATION OF NEED FOR CHANGE.*
2. *SUBMISSION OF CHANGE REQUEST.*
3. *REQUIREMENTS ANALYSIS.*
4. *APPROVAL/REJECTION OF THE CHANGE REQUEST.*
5. *SCHEDULING OF THE MAINTENANCE TASK.*
6. *DESIGN ANALYSIS.*
7. *DESIGN REVIEW.*
8. *CODE CHANGES AND DEBUGGING.*
9. *REVIEW OF THE PROPOSED CODE CHANGES.*
10. *TESTING.*
11. *DOCUMENTATION UPDATE.*
12. *STANDARDS AUDIT.*
13. *USER ACCEPTANCE.*
14. *POST INSTALLATION REVIEW OF CHANGES.*
15. *COMPLETION OF MAINTENANCE TASK.*

Figure 1-1: Phases in software maintenance process.

maintenance this phase should end up with a preliminary design for the affected part of the system. For all types of maintenance activities the requirements analysis phase should come up with an estimated cost of this activity and associated actions that are to be taken in order to incorporate the results of the maintenance activity into the system.

The scheduling decision about a maintenance activity should be based upon the operational requirements of the system at that particular time. In order to reach an intelligent decision on this matter, close cooperation with the user is imperative and it should be done by a Software Configuration Control Board (SCCB) that consists of user representatives, system engineers and maintenance team management (see [Ref. 11]). The scheduling should take into account the current backlog of Maintenance activities. It should be noted that some problems may have utmost importance for the system and may be assigned top priority in the working schedule. Thus, a scheduling decision for such a problem may cause rescheduling of maintenance activities that had been already assigned to programmers.

The implementation phases include the design analysis and review, code changes and their review, and testing of the change (preferably, in full scale mock-up of the operational environment). The success and cost of these phases depends very much on the quality and the results of the previous maintenance process phases, especially problem research. The implementation phases may repeat themselves until the correct solution to the required change has been found and implemented.

The phases following implementation also include updating all "programming" documentation related to the changed component of the system together with enforcement of programming and documentation standards. The actual updating of the programming documentation should be done in each and every step that results in some change to a system's design and/or implementation, i.e., after each step is done the appropriate documentation should reflect the incorporated changes. The significance of the separate documentation step in the maintenance process is the auditing of the updated documentation and enforcement of standards.

The user acceptance and post installation review phases have great importance. Their main purpose is to make sure that the system's capabilities are preserved (or enhanced), no new bugs are introduced and the requested change is incorporated into the system. These phases require close cooperation with the user and may involve updating user manuals.

D. MANAGING AND CONTROLLING THE MAINTENANCE PROCESS

As for the software development life cycle, management of the maintenance team must identify, control, and record the maintenance process by tracing all of its activities [Ref. 12]. Considering the software maintenance process explained above and taking into account the additional characteristics of maintenance in military computing systems, it is obvious that managing and controlling the software maintenance becomes a very complicated task. This task requires as much help as possible in order to make it feasible for an average human being.

The ultimate goal of software maintenance management is to keep all systems functioning in a correct and required way, and to respond to all user requests in a satisfactory manner. But taking into account the problems and complications of software maintenance discussed above, the more realistic goal of maintenance management is to keep the maintenance process under control [Ref. 2].

In order to achieve this goal, the maintenance manager must be able to review all pending change requests, plan for and schedule maintenance according to current priorities of the required changes. This means that the maintenance manager must process a large amount of information which must be formalized and accepted throughout the maintenance organization.

The pending requests for system's changes should be submitted formally to the maintenance management and should include justification of why that change should be made, and the priority for the implementation of this change to the system. The decision about requested change implementation must be formally recorded and it should include information

about estimated costs and implications of the change. Such a formal decision may serve as a trigger in change implementation, planning and scheduling.

Taking into account the importance of maintenance activity and the efforts and resources required, maintenance activities may be grouped together or be performed independently.

Performing maintenance activities on an independent basis has the advantage of bringing a quick solution to a problem in one part of the system, but it complicates the management's task of achieving the best possible utilization of human and machine resources. Such solutions (usually called "patches") should be simple and quick to apply, and they should be used with care in order to solve some high priority problems (like fixing a bug that blocks the use of an important system's feature). Usually, after introducing a number of such "quick fixes", i.e., "patches", some "clean-up" maintenance activity that will incorporate these fixes into the "regular" code, should be performed. An independent maintenance activity (usually) does not introduce new system wide features, it involves a limited number of modules, and sometimes does not require changes to the user procedures and manuals.

Grouping the maintenance activities together creates (in effect) a new, upgraded edition of the system that (usually, but not necessarily) differs from a previous version by additional system-wide features. This usually implies an update of the user's operating procedures and manuals. Grouping of the maintenance activities allows better planning and utilization of the available resources for the maintenance team management and the organization as a whole and is a preferable practice[2]. Such grouping requires an information base of all the maintenance activities to be performed and a lot of planning on the part of the management. In order to perform such planning, the management must have at its disposal information about job assignments for each task, its current status (completed, at work, not started, etc.), estimated time to accomplish it, amount of the time spent so far, scheduling constraints, deadlines (if

---

[2] Such grouping of maintenance tasks is sometimes called "scheduled maintenance" and is in favor with some commercial organization's DP departments as maintenance policy.

any), and dependencies upon other tasks. Another type of information that is important for planning is data about working personnel and programmers potential and current workload.

Due to the fact that every maintenance activity is based upon the current state of the system, which in turn is based upon the previous states and maintenance tasks performed, for maximum cost effectiveness, management considerations and judgement should be based upon the information concerning the history of the system maintenance with the current state having the strongest, but not exclusive, influence [Ref. 4].

One of the important tools the maintenance management has at its disposal is a software configuration management and control system. This system is the most significant in coordinating the activities concerned with the software modules and exercising control over the system evolution [Ref. 13].

E. ENGINEERING DATABASE FOR SOFTWARE MAINTENANCE

During the maintenance activity a large amount of information is created and processed. This information includes bug reports and appropriate findings, user change requests and resulting actions, SCCB decisions, maintenance activity costs (estimates and actual values), etc. Since this information has great importance for future maintenance cycles, it should be recorded in a system maintenance journal (see [Ref. 14] and [Ref. 9]), and it should become an integral part of the system's documentation (in a broad sense).

To use the information in the maintenance journal efficiently, it must be organized as a database with online access and proper retrieval facilities. In this database bug reports and requests for system changes can be managed and controlled, and, additionally, every maintenance activity can be uniquely represented by its causes and reasons, as well as its scope and effect. This database can and should serve as an MIS framework for the maintenance process and provide a basis for tools oriented toward software maintenance management. These tools should provide a means for change control, software system configuration control, etc. In

order to do this efficiently, the engineering database should incorporate all the necessary information about the software configuration as well. This means that there should be a single and complete data repository that will serve as the engineering database for all maintenance activities.

The requirements for such an engineering database should include the following properties:

- The database should support the software maintenance process throughout its life cycle.
- The database must be the main repository for all the operational information concerning the maintenance process, i.e., for user requests for changes, requirements analysis results, SCCB decisions, etc.
- The database should allow centralized control over all maintenance activities and provide automatic actions and operations with the possibility of manual override.
- The database should contain all of the system's components configuration modules (each in his own format, i.e., text, compilable source, etc.) and it should support a full scale configuration management and control system.
- The database should reflect the inter-relationships between the software modules of the system and support code reusability and inheritance (as it may be required for benefits of Object Oriented Programming).
- The database should provide for automatic consistency checking (based upon the module constraints network) and provide means and operations for automatic derivation of executable modules.
- The database should provide the maintenance management with information about the ongoing state of the software maintenance and provide support for management decisions: planning, scheduling, task assignment, cost estimation, status checking, etc.
- The maintenance process history should reside within the scope of the engineering database and be available (online) to authorized personnel.
- For each type of maintenance action the database must support appropriate views of information and provide controlled concurrent access to relevant information for authorized personnel.
- The database management system should be flexible enough to allow initial configuration and re-configuration for changes in software maintenance organization.

In order to implement an engineering database that adheres to the above requirements, an appropriate mathematical model that ties together the software configuration information and

the maintenance managerial information should be established. Only after creation of such a model can an effective and efficient implementation of the database and appropriate tools be done.

The next chapter of this work presents a Model of Software Maintenance that integrates the relevant configurational and managerial aspects of the maintenance process management into one coherent framework. This Model is intended to serve as the theoretical basis for a maintenance-oriented engineering database that will address the requirements described above.

II. THE MODEL OF SOFTWARE MAINTENANCE

The main objective of the Model of Software Maintenance is to provide a framework that integrates information about software maintenance activities with configuration control. The model is not concerned with the mechanics and the details of the maintenance programmer task and it assumes organizational paradigms that comply with ANSI/IEEE Guide to Software Configuration Management [Ref. 11] as follows:

- The management of the software maintenance organization exercises a formal type of change control, i.e., the system configuration changes only as a result of a maintenance action authorized by the management.
- The software configuration management system is used as a tool to coordinate maintenance activities that occur within the context of the system, and the implementation of the control is done utilizing software libraries.
- All of the verified software objects are contained in a controlled software library (i.e., master library) that is under direct control of the maintenance management, i.e., all changes to components of the master library must be authorized.
- The actual programming work is done using the dynamic (programmer's) library which is outside the master library, i.e., when each programmer is assigned to perform a maintenance activity appropriate software objects are copied from the master library to the dynamic one, and the programmer has free access to them; final results of his work are transferred from the dynamic library to the master library when his work has been tested, verified and accepted.
- The products of the configuration (e.g., executable software objects) are derived from the system's configuration repository and installed at the "production" site (i.e., "outside" the configuration repository). These software products are considered to be the "exports" of the configuration.
- Since product derivation may be required at any point of time, the system's configuration must be consistent at all times, i.e., at no time may derivation of executable objects be compromised because of consistency problems of existing completed software objects.

Such organizational paradigms are common to most software development and maintenance organizations that deal with software systems of large and medium size.

A. BASIC CONCEPTS AND UNDERLYING MODELS

The Model of Software Maintenance is based upon the ideas and notations of E. Borison's Model of Software Manufacture [Ref. 15], the Graph Transformation Model for Configuration Management Environments [Ref. 16] and the observation about relations between the maintenance activities and the configuration of the system.

1. Relation Between Maintenance and Configuration

A direct effect of the software maintenance activity is a change in one or more components of the system. These changes affect the configuration of the system, its semantics, and its functionality.

The relationship between the software configuration of the system and maintenance activities applied to it can be formulated as follows: each maintenance activity is a function on the power set of the system's software configurations; when applied to a subset of a system's configurations it results in an updated subset of the system's configurations. In a mathematical sense, the system software configuration is generated by the maintenance activities; for any change in the system software configuration there exists some maintenance activity that leads to its creation.

Therefore, we can model the evolution of the system during the maintenance phase of its lifecycle as a graph that consists of software objects that comprise the system configuration, and the maintenance tasks and activities that are applied to these objects. Such a graph captures the semantics of the above principles and allows creation of an abstract mathematical model that incorporates the specifics and necessary information of both software maintenance management and system configuration management and control into one coherent framework.

2. The Model of Software Manufacture

Borison's Model of Software Manufacture [Ref. 15] views software components as immutable objects, i.e., they can be created and destroyed, but once created their values cannot be modified. Any attempt to change such an immutable object creates a new version of this object that differs from the original one. Once created, the software components are not destroyed, they remain alive throughout the lifetime of the whole system and may be used later to spring off new genealogies.

A manufacturing activity (called a ***step*** in the model) is a derivation relationship between two sets of components: an input set and an output set (see Figure 2-1). In the original Model of Software Manufacturing the manufacturing step "works" on one or more inputs and "produces" one or more new components. Each invocation of a manufacturing step is considered distinct, whether or not it operates on different inputs. Both the manufacturing steps and the software components are given unique labels for the lifetime of the system in order to distinguish between them.

The model of Software Manufacture views the system as a finite, labeled, directed acyclic graph (G) of components (C nodes) and manufacturing steps (M nodes). The graph G is bipartite, i.e., manufacturing nodes alternate with component nodes.

The graph G is represented by a tuple <C,M,I,O> where C and M are sets of nodes and I and O are sets of edges:

- The set C represents all software components of the system.
- The set M represents the manufacturing steps applied to the components of the system.
- The set I represents the input relations between components and the manufacturing steps.
- The set O represents the output relations between the manufacturing steps and the components.

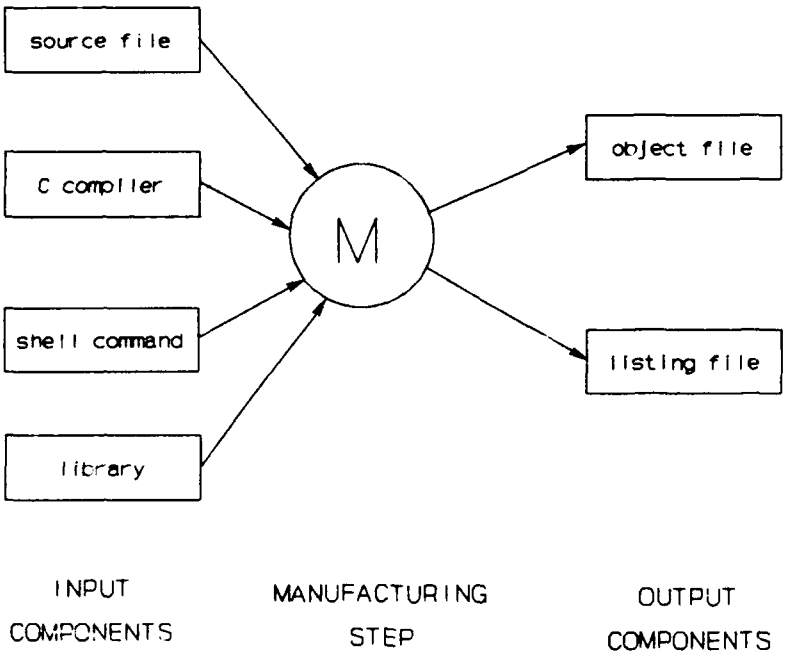


Figure 2-1: A Step in the Model of Software Manufacture.

Since no component can be a product of more than one manufacturing step, the set **O** is restricted so that:

(2-1) $\forall\ M_i,M_j \in \mathbf{M}$, If $\exists\ C \in \mathbf{C}$ such that $(M_i,C) \in \mathbf{O}$ and $(M_j,C) \in \mathbf{O}$, then $M_i = M_j$

Also, a set of primitive components (i.e., the primitive configuration, the one that is used to set up the system) can be defined as follows:

(2-2) $\mathbf{P} = \{\ C \in \mathbf{C} \mid \neg\exists\ M \in \mathbf{M}$ such that $(M,C) \in \mathbf{O}\}$

Let $\mathbf{D}^* = (\mathbf{I} \cup \mathbf{O})^*$ be the reflexive transitive closure of the union of the input and output relations **I** and **O**, then following properties can be stated:

- The inter-component and inter-manufacturing step dependencies are defined as follows:

(2-3) Component C_j depends on component C_i <=> $(C_i,C_j) \in \mathbf{D}^*$

(2-4) Step M_j depends on another step M_i <=>$(M_i,M_j) \in \mathbf{D}^*$

- For any component C the set of manufacturing steps that are affected by a change in C is defined as follows:

(2-5) $\mathbf{M_C} = \{\ M \in \mathbf{M} \mid (C,M) \in \mathbf{D}^*\ \}$

A configuration in the Model of Software Manufacturing is defined as a tuple <G,E,L> where G is the graph described earlier, $E \subseteq C$ is a set of components designated as exports of the configuration (i.e., components that are designated for a use outside of the configuration), and L is a labeling function that distinguishes different components of the system. The graph G contains only those manufacturing steps that are necessary to produce an export configuration of the system, i.e., the following holds true:

(2-6) $\forall M \in M \; \exists C \in E$ such that $(M,C) \in D^*$

The original Model of Software Manufacture presented briefly above is too general, oriented towards use of tools and application of derivation transformations to some components in order to create others, and is concerned only with manufacturing steps that result in export components. Also, the components in the Model of Software Manufacture are not limited to conventional software modules (e.g., source code files) and actual parameters for tool invocations are considered to be "legal" components of the model. The manufacturing steps have no concrete existence, they are taken to be the derivation relations between inputs and outputs.

The Model of Software Manufacture is not suited for the specifics of maintenance tasks, and it must be refined in order to serve as a model for software maintenance process.

3. Relations Between Software Components

The Graph Transform Model [Ref. 16], classifies software objects into two categories: ***re-derivable*** and ***non-re-derivable***. Re-derivable objects can be automatically reconstructed by applying some tool to some set of software objects. All other objects are considered non-re-derivable (e.g., "source" objects). The software objects may have attributes, which can specify computational procedures that should be applied to the components in order to perform specific transformations.

There exist two important relations between non-re-derivable and re-derivable objects: ***is-component-of*** and ***derives***. These relations have a direction and are easily modelled using digraphs (see Figure 2-2).

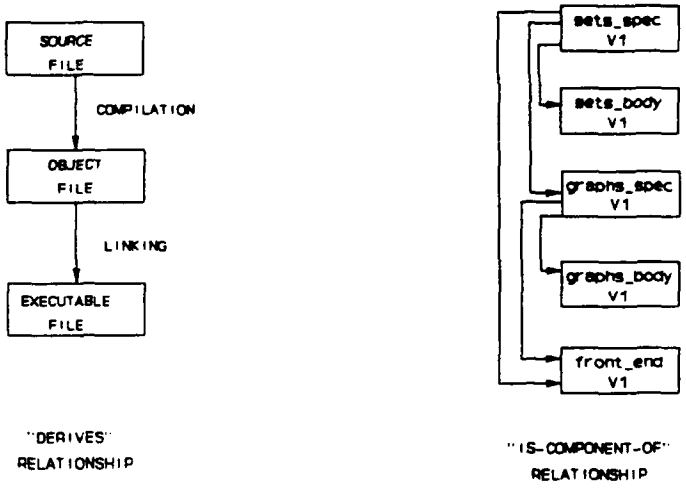


Figure 2-2: Example of Relations Between Software Objects.

The relation "derives" is defined between non-re-derivable and re-derivable objects and it represents a transformation of one or more software objects into another (e.g., compilation of source modules into linkable object modules). The "derives" transformations are typed transformations that are applied to objects of a specific type. These transformations are very specific, they are known a-priori, and can be applied automatically using the information about the type of the software object and the attributes of the object itself. The "derive" transformations are associated with the use of software tools in the process of programming, and they usually are invisible to the management or users of the system.

The "is-component-of" relation is defined between non-re-derivable objects only and it represents the use of one component by another component of the system (e.g., use of packages in the ADA programming language). To denote the "is-component-of" relation we will use a convention in which the "is-component-of" relation between components C_i and C_j means that C_i is a component of C_j.

The "is-component-of" relations between components are defined by the software system overall design and module decomposition. These relations may be specified in the component itself (e.g., using compiler directives in programming languages - "#include" in C, "with" in ADA, "COPY" in some COBOL dialects) or explicitly stated as attributes representing additional information required for deriving transformations (e.g., library specifications in linking commands). In both cases the relation information is defined a-priori, and is stable relative to the dynamics of the software changes due to the maintenance process. These stable relationships may change as a result of maintenance, but these changes are few and far in between compared to the changes in the components themselves.

B. CONFIGURATION GRAPH OF THE MAINTENANCE MODEL

The Model of Software Maintenance is comprised of two basic elements: ***system components*** and ***maintenance steps***.

The system components are immutable and non-re-derivable software objects. The system components of the Model of Software Maintenance correspond to the components in the Model of Software Manufacture, with the exception that the components must have concrete existence as software objects of the system. Programming tools and their invocation parameters are not considered to be system components in the model. System components are henceforth called components.

The maintenance steps correspond to manufacturing steps of the Model of Software Manufacture with the following differences:

- A maintenance step is a "representation" of the organizational activity concerned with initiation, analysis and implementation of one request for a change in the system.
- A maintenance step may be atomic (i.e., applied to at most one system component and produces at most one output component), or be composed from a number of atomic steps.
- A Model of Software Maintenance allows for the existence of empty steps that do not produce output components.

- The model also allows for existence of "dead moves" (i.e., steps that do not lead to production of "useful" components). The existence of such steps is motivated by the need to keep correct records of all maintenance activities, including those that have taken a "wrong turn".
- Deriving transformations are not considered to be maintenance steps and are not represented in the Model.

Additionally in the Model of Software Maintenance, for each maintenance step a ***scope of a change*** is defined as all sub-systems (or systems[3]) to which the step is applied.

The system configuration is an acyclic directed graph (digraph) G of components (C nodes) and maintenance steps (M nodes), in which the components and steps are connected by two relations: inputs (I arcs) and outputs of the maintenance steps (O arcs). The output relations of maintenance steps are defined between a maintenance step and the non-re-derivable component it produces. The input relations are defined between a maintenance step and the system components which are necessary to produce an output component that is consistent with the rest of the system. Naturally, no component that has an output relation with a maintenance step can have an input relation with the same step, i.e., no input and output "feedback" relations are allowed. This extends to any path of relations, thus avoiding cycles and complying with the requirement that G is acyclic. It should be noted that input and output relationships between components and atomic maintenance steps define a bipartite digraph.

We will represent the input and output relations by sets of the components for which the relations hold. We will use notation in which for a maintenance step M_q (where q stands for the step's label, e.g., an index in an enumeration) its input and output sets are denoted I_{M_q} and O_{M_q} (respectively). I_{M_q} and O_{M_q} are sets of system components which have input and output relations with the step M_q, respectively (see Figure 2-3).

---

[3] The exact definition of a system will be provided later.

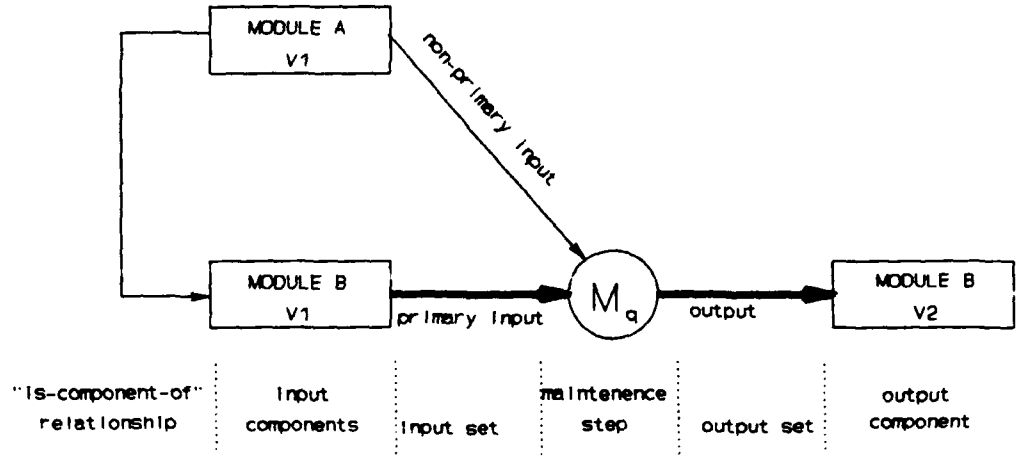


Figure 2-3: An example of a Maintenance step.

We can formalize some of the basic principles and definitions of the Model of Software Maintenance as follows:

(2-7) $\forall$ maintenance steps M, $|I_M| \geq 0$ and $|O_M| \leq 1$.

(2-8) M is an empty step iff $O_M = I_M = \emptyset$.

(2-9) For all maintenance steps M, if $I_M \neq \emptyset$ then $|O_M| = 1$.

(2-10) If component $C \in O_{M_q}$ then $C \notin I_{M_q}$.

(2-11) $\forall$ $M_i, M_j \in M$, If $\exists$ $C \in C$ such that $(M_i,C) \in O$ and $(M_j,C) \in O$, then $M_i = M_j$.

(2-12) $P = \{ C \in C \mid \neg\exists M \in M$ such that $(M,C) \in O\}$

(2-13) Let $D^* = (I \cup O)^*$ be the transitive closure of the union of the input and output relations I and O, then

a) Component C_j depends on component C_i <=> $(C_i,C_j) \in D^*$;

b) Step M_j depends on another step M_i <=> $(M_i,M_j) \in D^*$;

c) Set of steps affected by a change in component C $M_C = \{M \in M \mid (C,M) \in D^*\}$

It should be noted that the properties (2-11) - (2-13) of the Model of Software Maintenance are similar to properties (2-1) - (2-5) of the original Borison's Model of Software Manufacture, upon which the Model of Software Maintenance is based.

C. MAINTENANCE STEP STATES

During the execution of the maintenance process the maintenance activity, which corresponds to a step in our model, can be in several possible states (see description of the maintenance process in previous chapter). These states represent some of the dynamic aspects of the maintenance process initiated as a result of a system change request.

For the purposes of the Model of Software Maintenance the following five states of a maintenance step are defined:

- Invoked.
- Pending.
- Implementing.
- Completed.
- Abandoned.

Each of the above states corresponds to several phases of the maintenance process described in [Ref. 2], and corresponding sub-states can be defined for each of the above states in the implementation of the model. For brevity we call a maintenance step by the name of the state it is in, e.g., "pending step", "implementing step", etc.

Transition of a maintenance step from one state to another (see Figure 2-4) is performed as a result of an explicit decision made by maintenance organization management. By controlling the states of the maintenance steps, the maintenance management exercises direct control over both the software maintenance process and the system configuration.

In the "invoked" state the maintenance step is created according to a requirement for a change in the system. In this state the originated change undergoes analysis which estimates the resources required for the step's implementation, designates inputs and defines the scope of the step. At this stage the maintenance activity is not yet approved for implementation, and it is not yet linked to any component of the system. If the maintenance activity is not approved by SCCB, then the maintenance step state becomes "abandoned".

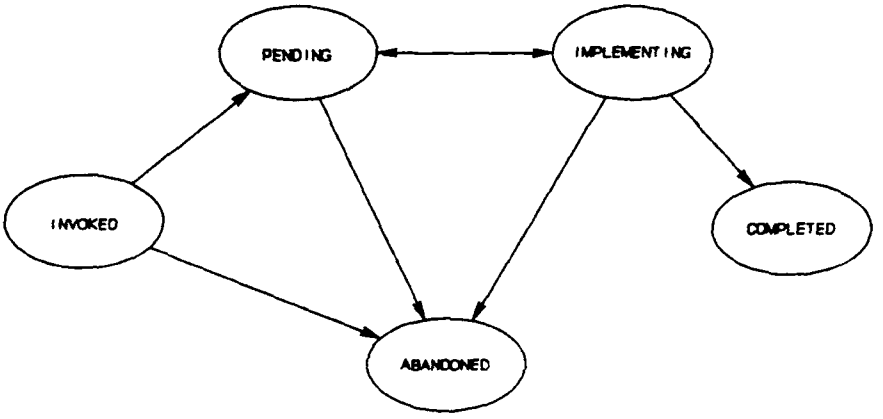


Figure 2-4: Maintenance Step States and State Transitions.

In the "pending" state the maintenance step is approved but it has not yet started its implementation phase. In order to be implemented, the pending maintenance step must be scheduled and assigned to a programmer. Scheduling of a maintenance step should resolve all possible inconsistencies that may arise as a result of concurrent implementation of the pending steps. A maintenance step in the "pending" state can be "abandoned" (i.e., transferred to "abandoned" state) and forgotten there.

In the "implementing" state the actual implementation and testing of the requested change is performed. At the transition to this state from the "pending" state, the binding of input components occurs, and the output component is created. At this stage, the output component is a placeholder for the future component contents of the maintenance step results, i.e., the output component is "empty" at the beginning, and its contents are produced during the implementing phase of the maintenance step. It should be noted that this behavior does not compromise the immutability of the components, since the immutability feature is applied after the component is created, i.e., after the "implementing" state has been completed. Also, note that the "implementing" state can be "rolled back" into the "pending" or "abandoned" state, and

in such a case the output of the step is invalidated and is no longer a component of the system.

The "implementing" state changes into the "completed" state after the requested change implementation is done and tested. At the transition to this state the output component's contents are frozen and entered into the system component repository and it becomes an approved member of the system. Then the step itself terminates and becomes part of the permanent record of the system.

The "abandoned" state is the final state for all maintenance steps that were not approved by the SCCB or "killed" by the management in the "pending" and "implementing" states.

It should be noted (see Figure 2-4) that only the maintenance steps in the "implementing" state can be "rolled back" into "pending" state. By doing so all the work that had been performed to implement this maintenance task may be lost due to later changes in the system that may affect the "rolled back" step. Therefore such decisions should be made with insight and great care.

D. MAINTENANCE STEP INPUTS

The nature of an atomic maintenance step is to incorporate a single change in a single component of the system. We will call such a component a ***primary input*** of the atomic maintenance step (see 3), and we will limit the number of primary inputs for an atomic maintenance step so, that:

(2-14) $\forall$ maintenance step M $\exists$ at most one primary input $C \in I_M$.

In order to capture the semantics of dependencies of some system components on other components (as defined in (2-13)), we will introduce the notion of ***non-primary inputs*** of a

maintenance step. The non-primary inputs of the maintenance step M belong to I_M, the input set of this step, and they are defined by its result:

(2-15) For a maintenance step M_q the set I_{M_q} consists of the primary input and all other components that are used in creation of the component $C \in O_{M_q}$ and in the deriving transformations that produce other software objects from it.

It immediately follows from (2-15) that:

(2-16) If $\exists$ an "is-component-of" relation between C_j and $C_i \in O_{M_q}$ then $C_j \in I_{M_q}$.

The non-primary inputs may be determined manually or automatically using knowledge based techniques. If the primary input C_i of a maintenance step M_q is a primitive component (i.e., $C_i \in P$ as defined in (2-12)), then the non-primary inputs of M_q can be computed from the "is-component-of" relations of system components with the component C_i (see Figure 2-3). Otherwise the inputs of the previous step M_p with $C_i \in O_{M_p}$ may be used to compute the set I_{M_q}. Facilities for updating the input set of a maintenance step in the pending and implementing states must be provided in the implementation of the Model.

It should be noted that the existence of the inputs for the maintenance step is not a necessary condition (see (2-7)). There may exist a maintenance step without inputs that produces an output component. Also, there may exist a maintenance step with non-primary inputs only that produces an output component. An example for a such case is the creation of a new component by merging the contents of a number of existing components, although it should be noted that there exists another way to represent such merging: one component can be the primary input of the maintenance step that performs the merging, while other components that are used in the merging process, become the non-primary inputs of the merging step.

E. DESCENDENCE RELATION, GENEALOGY TREES AND SYSTEMS

We will define a descendence relation by saying that primary input C_i of a maintenance step M is an ***ancestor*** of the resulting component $C_j \in O_M$, and that the component $C_j \in O_M$ is a ***descendent*** of C_i.

Descendence is a transitive relation, i.e.,:

(2-17) Component $C_k \in O_{M_p}$ is a descendent of C_i iff C_i is the primary input of a step M_p or the primary input of M_p is a descendent of C_i.

In order to distinguish between types of descendence relations for later use, we will call the non-recursive descendence a direct descendence.

The descendence relation defines ***evolution genealogy*** (called genealogy for shorthand) subgraphs of the configuration graph G. These subgraphs are created using only the primary input and output relations of the maintenance steps. Because the graph G is acyclic, and due to the restriction of at most one primary input for a maintenance step, each element of the genealogy subgraph is a tree. Thus, the following hold true:

(2-18) All descendants of a component C belong to the genealogy tree T that has C as its root or one of its nodes.

(2-19) There exists a unique path between component C and any of its descendants.

We call a genealogy tree with a component C as its root a C genealogy tree and we will denote it as T_C. A genealogy tree can be a subtree of other genealogy tree(s), or be a spanning tree, which is defined as follows:

(2-20) T_i is a spanning tree $\Leftrightarrow \neg\exists\ T_j$, $i \neq j$ such that $T_i \subset T_j$.

Using the notion of genealogy trees we can define software ***systems*** (or products), and we will say that a software system S is uniquely identified by a set of genealogy trees that comprise it, i.e., $S = \{T_i\}$. It should be noted that the whole system is defined by a set of all spanning genealogy trees, and in the case of a sub-system, the genealogy trees T_i are not necessary spanning. Since sub-systems themselves can be viewed as systems, we will not

distinguish between them unless it is required. The following rules apply to both systems and sub-systems alike, since software products can be viewed as either systems or sub-systems.

We will say that software system S is ***complete*** if the following holds true:

(2-21) $\forall\ T_i \in S$, if $\exists\ T_j \mid C_k \in T_i$ depends on $C_n \in T_j$ or C_n is a component of C_k, then the tree $T_j \in S$.

We can express the relations between components and systems as follows:

(2-22) Component $C_i \in S \Leftrightarrow \exists\ T_k \mid C_i \in T_k$ and $T_k \in S$.

We also can define the scope of a change for a maintenance step (denoted as K_{M_q}) that, as mentioned earlier in this chapter, represents software (sub)systems to which the maintenance step is applied:

(2-23) if K_{M_q} is a scope of step M_q with primary input C_i then $K_{M_q} \subseteq \cup S_j \mid C_i \in S_j$.

The inclusion $K_{M_q} \subseteq \cup S_j$ in the above rule means that there may be a situation in which (by a virtue of a maintenance management decision) an influence of a maintenance step will be limited to particular (sub)systems.

F. DESIGNATING PRIMARY INPUT FOR A MAINTENANCE STEP

Designating a component C as an input to a maintenance step means that this step will take the component C or one of its descendants as a primary input. Such a designation is possible only for pending maintenance steps, and it does not prevent the use of this component by other maintenance steps, i.e., it does not "lock" the input component.

The actual binding of primary input with the designated component can be performed only if the input component is ***available*** and it is done at the transition time of the maintenance step from the "pending" state to the "implementing" state, after the step is scheduled and assigned to a programmer. After the primary input is bound to the implementing maintenance step, its non-primary inputs are determined.

The primary input component is available for a maintenance step if it is a primitive component or it was created by some completed maintenance step, i.e., it exists and is not currently being worked on:

(2-24) A component C is available iff $C \in P$ or $\exists$ completed M such that $C \in O_M$.

Components that are currently under work by some uncompleted maintenance step may be used as non-primary inputs of a step, and they are considered available the moment they are created by an implementing maintenance step.

The designation of a primary input component to a maintenance step can be ***specific*** or ***generic***. When specific designation is done, the maintenance step will be applied to some specific component that already exists in the system, e.g., to a specific version of the primitive component. There is no possibility of specifically assigning a component that is not produced yet.

In the case of a generic designation of component C, the maintenance step will be applied to the available descendent of C. As mentioned earlier, the actual binding of the primary input to a component that belongs to the genealogy tree of C will take place when the maintenance step begins its implementation phase. For example, in the case of a generic designation, if there are several maintenance steps with the same designated input component C, then the first maintenance step is applied to the component C, the second is applied to the descendent of C, and so on.

It should be noted that, because of the possibility of creating "parallel" genealogies that originate from the component C, the generic designation may not be unique.

G. DECOMPOSITION OF MAINTENANCE TASKS

Sometimes, after analysis of a requested change that initiates a maintenance step, it becomes apparent that the implementation of the original change leads to changes in several system components. The original request for a change is represented as a maintenance step.

However, because a *maintenance step* can be applied to at most one primary input and produces at most one output component, the maintenance step invoked by the original request for change becomes a composite maintenance step; it "spawns" a number of new atomic *maintenance steps. In such cases, it is important* to record a relation between the composite step and the steps that are "spawned" by it. We will call this process "***step decomposition***", the relation "***spawned***", the composite step a "***spawning step***" and the newly invoked steps "***spawned***". The step decomposition process is recursive, i.e., spawned steps may themselves be composite maintenance steps.

In order to eliminate the possibility of unnecessary relations between composite steps and their spawned steps, the composite maintenance step may not produce any new component by itself, i.e., it is an empty step.

The composite step decomposition takes place when the step's implementation is authorized and it creates a set of spawned steps. These spawned maintenance steps are created directly in the "pending" state, i.e., they are considered to be authorized by the virtue of authorizing the composite step. The spawned steps behave normally, meaning that they do not differ from non-spawned maintenance steps and they may have relations with one another or some other maintenance steps. Because the spawning step is an empty step, there are no dependency relations between the spawned steps and their spawning steps.

In order to provide consistency in treating composite and atomic steps, the following constraints are imposed on some state transitions of the spawning and the spawned steps:

(2-25) The spawning step is transformed automatically from pending to implementing states when one of its spawned steps performs this transition.

(2-26) The spawning step performs an automatic transition from implementing to completed state when all of its non-abandoned spawned steps have done so.

(2-27) Abandoning a spawning step will automatically abandon all of its spawned steps.

(2-28) The transition of a spawning step to abandoned state is done automatically when all of its spawned steps are abandoned.

It should be noted that, since all spawned steps are intended to implement a specific maintenance task, a single maintenance step (atomic or composed alike) can be spawned directly by only one composite maintenance step, i.e., the graph formed by a spawning relationship is a tree.

H. INDUCED MAINTENANCE STEPS

An engineering change in a key component of a software system may compromise the consistency of systems that belong to the scope of the step by affecting other components of these systems in such way that some action is required in order to keep them consistent. For example: a change in the specification of some ADA package requires some action to be performed on all other components that use this specific package before any new software product can be successfully derived.

We will define an ***induced maintenance step*** as a step that must be performed in order to keep the system's consistency due to a result of another maintenance step. The importance of induced maintenance steps is in alerting the maintainers and the management to changes in key modules of the software product and enforcing constraints on performing any uncoordinated maintenance step on the affected components.

It should be noted that a change in one component may trigger a change in another, which may in turn trigger a change in third component, and so on, i.e., the changes may be triggered recursively. We will call a component that originated the change propagation a ***triggering*** component, and the step that uses it as its primary input an ***inducing step***.

Additionally, the propagation of the changes triggered by an inducing step must be restricted to the scope of the inducing step.

In order to define specifically the relevant maintenance steps that are affected by a change in component C, we will introduce the concepts of *latest descendent* and *latest*

maintenance step. The latest descendent of a primitive component is a component that is not used as primary input by any maintenance step, i.e.,:

(2-29) C_i is a latest descendent of C_j iff C_i is a descendent of C_j and $\neg\exists$ M_q such that C_i is the primary input of M_q.

The latest maintenance step is defined as follows:

(2-30) Step M is the latest step with respect to C_j iff the component $C_i \in O_M$ is latest descendent of C_j.

We refine the original definition of the set M_C (see (2-13)) using the notion of the latest maintenance step as follows:

(2-31) M_{Cj}, the set of maintenance steps affected by a change in triggering component C_j, consists of latest steps M which have C_j as their non-primary input and O_M belongs to the scope of an inducing step that implements the change in C_j.

Since a change in one system component may lead to the inconsistencies with the primitive components (i.e., the components that were not produced by any maintenance step), the above definition is not sufficient and we will introduce the notion of an ***affected component***. A system component, whose consistency with the rest of the system is affected by a change in some other component, is called an affected component and the following holds:

(2-32) A component C_i is affected by a change in C_j iff C_i belongs to a scope of an inducing step that implements the change in C_j and either $C_i \in O_M$ where step $M \in M_{Cj}$ or both $C_i \in P$ and $C_j \in P$ and C_j is a component of C_i.

Analogous to the definition of the set M_{Cj} (see (2-31) above), we can define a set of all components that are affected by a change in a component C_j (noted as C_{Cj}), using the recursive nature of the propagation of a change, as follows:

(2-33) A component C belongs to the set C_{Cj} if C is affected by a change in C_j or C is affected by a change in $C_i \in C_{Cj}$.

If $C_j \in I_{Mp}$ is a primary input and the set C_{Cj} is not empty, then the inducing step M_p induces maintenance steps $M_{p(n)}$[4], where n=1,2,..,$|C_{Cj}|$. An induced maintenance step $M_{p(n)}$ takes

[4] The notation $M_{p(n)}$ is used as a labeling for induced maintenance steps.

an affected component $C \in C_{Cj}$ as its primary input, and produces as its output a new component which is consistent with the direct descendent of the component C_j.

Uncoordinated propagation of the induced steps may cause the transient state of the systems configuration to be inconsistent. For example, a change of the package specifications without coordinated change of the package body may cause some incompatibilities and compromises the consistency of the whole system.

In order to keep the system configuration consistent, the inducing maintenance step together with its induced steps are performed as an atomic step, i.e., an inducing step and all of its induced steps (including those that were created recursively) should appear to perform their transitions from "pending" to "implementing" states and from "implementing" to "completed" states simultaneously. The following rules impose a semi-atomic behavior of inducing/induced steps by introducing the necessary constraints:

(2-34) An inducing step M_q with primary input C_j can start its implementation phase iff all steps $M_p \in M_{Cj}$ are completed.

(2-35) An induced step $M_{q(n)}$ with primary input $C_i \in C_{Cj}$ can start its implementation phase iff the inducing step M_q with primary input C_j has already done so.

(2-36) An inducing step M_q with primary input C_j can become completed iff all induced steps $M_{q(n)}$ with primary input $C_i \in C_{Cj}$ are completed.

(2-37) Any "roll back" transition of the inducing step causes the same transition to be performed on all its induced steps.

(2-38) An induced step can be "rolled back" only by "rolling back" all of its inducing steps.

(2-39) Abandoning an inducing maintenance step causes all of its induced steps to be abandoned.

(2-40) An induced step can be abandoned only by abandoning its inducing step.

The meaning of rule (2-34) is that the inducing step cannot begin its implementation before it assures that all steps that it affects (see (2-31)) are already completed, and the primary inputs for its induced steps are available (see (2-24)). Rules (2-35) and (2-36) mean that the induced maintenance steps cannot begin their implementation before their inducing maintenance

step, and the inducing step cannot complete the implementation phase before its induced steps. Other rules mean that the induced step has no reason to exist by itself, without its inducing step.

Because of the dynamic nature of the system's configuration, the influence of an inducing maintenance step M_q with primary input C_j (i.e., the contents of the set C_{Cj}) may vary. The contents of the set C_{Cj} become static as a result of scheduling of the step M_q. Since the induced steps $M_{q(a)}$ depend directly upon the contents of C_{Cj}, they are invoked immediately after the inducing step M_q is scheduled for execution.

I. EXAMPLE OF MAINTENANCE STEP DYNAMICS

A simplified example of maintenance step dynamics is presented in Figure 2-5. The example represents maintenance activities applied to a system that consists of five components: **sets_spec**, **sets_body**, **graphs_spec**, **graphs_body**, and **front_end**. The system in the example represents graphs using sets, performs operations on them and presents the results to the user through procedures in the **front_end** module.

The example shows influence of a simple maintenance step on the system configuration, creation of distinct evolution genealogies, and creation and propagation of the induced maintenance steps.

The set of primitive components of the system consists of the following five modules:
P = {**sets_spec**.V1, **sets_body**.V1, **graphs_spec**.V1, **graphs_body**.V1, **front_end**.V1}

The "is-component-of" dependencies of the primitive components are defined and shown on the left side of the primitive components in the figure.

In the example, a request for a change in one of the **front_end** procedures initiates a maintenance step M_1 with the designated primary input **front_end**.V1. After the step is scheduled and implemented, it creates the new component **front_end**.V2, which is a direct descendent of the **front_end**.V1 component.

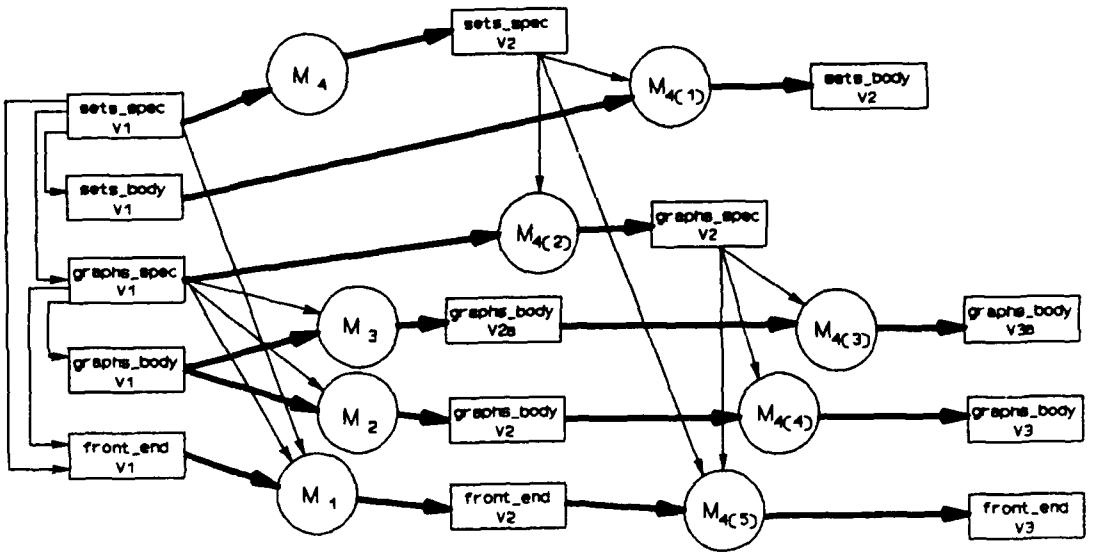


Figure 2-5: Example of Maintenance Steps Dynamics.

The non-primary inputs of the step M_1 are determined using "is-component-of" relation and they consist of components **graphs_spec**.V1 and **sets_spec**.V1.

A request for a change in the component **graphs_body**.V1 initiates maintenance step M_2, which results in the production of a new component **graphs_body**.V2. Another request for a change that corresponds to the maintenance step M_3 has a component **graphs_body**.V1 as its specific designated input and it results in a component **graphs_body**.V2a, giving rise to an evolution genealogy which is "parallel" to the one created by the maintenance step M_2.

The component **sets_spec**.V1 is used as a non-primary input in a number of maintenance steps, and a change in it will affect the following set of components:

$C_{sets\_spec.V1}$={**sets_body**.V1, **graphs_spec**.V1, **graphs_body**.V2, **graphs_body**.V2a, **front_end**.V2}

This set is computed recursively, starting with the set of components that are directly affected by **sets_spec**.V1 (i.e., the set {**sets_body**.V1, **graphs_spec**.V1, **front_end**.V2}), and then for each component in the set adding the components which it affects.

A maintenance step M_4 that implements a change in the **sets_spec**.V1 component induces the following maintenance steps:

- Step $M_{4(1)}$ that produces component **sets_body**.V2.
- Step $M_{4(2)}$ that produces component **graphs_spec**.V2.
- Step $M_{4(3)}$ that produces component **graphs_body**.V3a.
- Step $M_{4(4)}$ that produces component **graphs_body**.V3.
- Step $M_{4(5)}$ that produces component **front_end**.V3.

The step M_4 is completed only after all the maintenance steps it had induced (i.e.. $M_{4(1)}$..$M_{4(5)}$) are completed. Thus, the implementation of the inducing and induced maintenance steps is performed atomically, keeping the whole system consistent, i.e., any software object derived from the system components will be consistent with the latest changes incorporated in the system.

J. PRIORITY AND PRECEDENCE OF MAINTENANCE STEPS

During the life cycle of the software system, constraints that reflect the urgency and the partial ordering of the maintenance tasks arise from real life situations. These constraints influence the process of software maintenance, and they must be represented in the Model of Software Maintenance in order for the latter to be a realistic model.

We will represent the urgency of the maintenance tasks by assigning a small positive integer value as a ***priority*** value to each maintenance step that is needed to implement the task. The priority values of the maintenance step represent the relative urgency of the maintenance tasks, and they suggest an implementation ordering of the maintenance steps.

The priorities are assigned manually to the maintenance steps by an appropriate forum that includes the user (or his representatives) and the maintenance team management (e.g., SCCB), and they may be changed during the maintenance process according to the state of the system maintenance and external constraints. The process of assigning priority values to

maintenance tasks may use different methods and algorithms and is external to the Model of Software Maintenance itself.

Since the priorities are assigned to the maintenance steps according to the maintenance task, the following property should be preserved:

(2-41) If maintenance steps M_q and M_p are intended to implement parts of the same maintenance task, then steps M_q and M_p are assigned the same priority value.

In the case of assigning priorities to composite or inducing maintenance steps, the following defines the priorities of the spawned and the induced steps:

(2-42) If composed/inducing step M_q is assigned a priority value N, then all maintenance steps M_p that are spawned/induced by the step M_q are assigned the same priority value N.

The priority mechanism should not be misused, e.g., all maintenance steps should not be assigned the same priority. It is advisable to keep the range of the priority values as small as possible without altering their meaning.

In addition to the partial ordering that arises from assigning the priority values to maintenance steps, there may exist additional constraints that impose ordering constraints between two or more steps. These constraints may represent specifics of a maintenance task or inter-step dependencies that cannot be expressed by the input and output relations of the maintenance steps as defined in (2-4) and (2-13). The intent of such ordering constraints is to impose a sequential rather than a concurrent execution of the maintenance steps.

To account for the execution ordering ranking, we will introduce the "***precedes***" relation which is defined between pending atomic maintenance steps as follows:

(2-43) If atomic step M_q precedes atomic step M_p, then the step M_q must be completed before the step M_p begins implementation.

Because of the semi-atomic nature of induced maintenance steps (see (2-34) - (2-40)), the following holds true:

(2-44) If inducing step M_q precedes (or is preceded by) step M_p, then all induced steps $M_{q(n)}$ precede (or are preceded by) the step M_p.

(2-45) If induced step $M_{q(a)}$ precedes (or is preceded by) step M_p, then its inducing step M_q precedes (or is preceded by) the step M_p.

(2-46) An inducing step M_q cannot precede its induced steps $M_{q(a)}$, and vice versa.

Naturally, similar constraints apply to composite steps and the steps that are spawned by them, i.e.,:

(2-47) If atomic step M_q precedes composite step M_p, then the step M_q precedes all the maintenance steps spawned (directly or recursively) by the step M_p.

(2-48) If composite step M_q precedes step M_p, then all steps spawned (directly or recursively) by the step M_q precede the step M_p.

The "precedes" relation is transitive, asymmetric and irreflexive, i.e.,:

(2-49) If step M_q precedes M_r and step M_r precedes M_p, then step M_q precedes step M_p.

(2-50) If step M_q precedes M_p, then step M_p cannot precede M_q.

(2-51) $\forall$ steps $M_p \in \mathbf{M}$, M_p cannot precede itself.

The transitive and the asymmetric properties of the "precedes" relation imply that the graph of the "precedes" relation is acyclic, i.e., the situation in which step M_q precedes M_r, step M_r precedes M_p, and step M_p precedes step M_q is impossible. Also, the situation in which a spawned maintenance step precedes its spawning step is illegal.

Unlike the priority values that suggest an implementation ordering, the "precedes" relation imposes a strict ordering between two or more maintenance steps, e.g., in (2-43) the step M_q will be implemented before the step M_p even if the priority of the step M_p is higher than that of the step M_q. Therefore the ordering imposed by the precedes relation is "stronger" then the one we would obtain using only the priority values of the maintenance steps.

The "precedes" relation must be consistent with the dependency relation between maintenance steps (as defined in (2-13)), since the dependencies that propagate through the primary inputs impose the "precedes" relation between the steps:

(2-52) If $\exists\ C_i \in C$ such that $C_i \in O_{Mq}$ and C_i is the primary input of step M_p, then the step M_q precedes step M_p.

Also, there exist implicit precedence relations that concern inducing maintenance steps (see (2-34)) which can be stated explicitly as follows:

(2-53) If $\exists$ inducing step M_q with primary input C_j, then $\forall$ steps $M_p \in M_{Cj}$ step M_p precedes maintenance step M_q.

Recall that the "precedes" relation is not concerned solely with step dependencies that propagate through inputs of a maintenance steps. It deals also with the constraints that are external to the system configuration.

The "precedes" relations between maintenance steps should be defined by the maintenance management, and it is their responsibility not to misuse this mechanism. An incorrect use of the "precedes" relation will lead to introduction of many unnecessary constraints in the scheduling of the maintenance steps. On the other hand, the correct use of the "precedes" relation should improve the effectiveness of the maintenance process.

III. UTILIZATION OF THE MODEL OF SOFTWARE MAINTENANCE

Chapter II presents a Model of Software Maintenance that integrates the configurational, behavioral and managerial aspects of software maintenance and provides a conceptual framework for implementation of a maintenance-oriented engineering database introduced in Chapter I.

The main purpose of this chapter is to show how the Model of Software Maintenance can be used for the benefit of the maintenance organization as a whole, and the maintenance management in particular. The intent is to show the usefulness of the Model of Software Maintenance as a basis for a decision support system oriented towards the management of the maintenance teams and organizations. Such a system should be able to provide the management (among others) with timely information and appropriate tools to support planning of the maintenance schedules, tracking them with verifying milestones, providing maintenance process status and progress reports, and locating trouble areas in the planned schedule before it gets out of hand [Ref. 17]. It should also allow the management to assign maintenance tasks to individuals and track their progress throughout the maintenance process. It should address the issues of system configuration management, change control and to support adequate communications between different participants of the maintenance process, i.e., system users, members of SCCB, management and programmers [Ref.18].

This chapter will address the utilization of the Model of Software Maintenance in following areas:

- Planning and controlling the maintenance process.
- Configuration management and control of the maintained software.
- Gathering and processing metrics and statistics about the maintenance process.

In some of the above areas complete and detailed algorithms that provide a specific way to use the Model and the corresponding information will be presented. In other areas a way to make use of the Model will be shown without going into details. The reason for this approach is the aspiration not to tie the Model of Software Maintenance at this time to particular methodologies used by software development and maintenance organizations.

The need for a maintenance-oriented engineering database was described by Swanson in 1976 [Ref. 9] who also defined the basic entities of such a database and a set of basic maintenance summary reports. For the purposes of this chapter, we will assume existence of a maintenance-oriented engineering database that is built upon the Model of Software Maintenance. Such a database contains all of the system's components, the information about the maintenance tasks and steps, and the relationships defined by the Model of Software Maintenance. We also assume that the entities of the engineering database (e.g., system components, maintenance steps) have all the necessary attributes as shown in Figure 3-1. These attributes define the minimal information required for the algorithms that will be presented later in the chapter and do not preclude definition of additional attributes.

It should be noted that some attributes of the basic entities defined by Swanson can be computed automatically, using the properties of the relationships between system components and maintenance steps defined by the Model of Software Maintenance. For example, the number of source lines changed during a maintenance activity can be determined by comparing the primary input component and the output component of a maintenance step.

A. PLANNING AND CONTROLLING THE MAINTENANCE PROCESS

As explained in Chapter I, the software maintenance process is difficult to plan or predict due to its inherent complexities and characteristics. The Model of Software Maintenance allows an integrated approach to the problem of maintenance planning and leads to the definition of relatively simple algorithms that provide decision support for the management of the

Component Database Entity Attributes:

- *Unique component ID.*
- *Date of component creation.*
- *Component contents.*
- *Type of programming language used to create the component.*
- *Component size (e.g., in Lines of Code (LOC)).*

Maintenance Step Entity Attributes:

- *Unique maintenance step ID.*
- *Date of maintenance step creation.*
- *Software change requirement description.*
- *SCCB decision concerning the maintenance step.*
- *Maintenance step description.*
- *Programmer assigned to implement the maintenance step.*
- *Estimated execution effort for a maintenance step.*
- *Actual effort spent on execution of a maintenance step.*
- *Maintenance step execution starting time.*
- *Maintenance step execution completion time.*

Figure 3-1: Required Attributes for Engineering Database Entities.

maintenance teams and organizations. These algorithms can help managers plan the required size and expertise of the maintenance team, predict completion of a specific maintenance task, estimate the time to clear the backlog of maintenance request given a team of programmers, etc.

This section will address maintenance step scheduling, programmer's job assignment, maintenance process simulation and process execution control aspects of maintenance process planning and control.

1. Scheduling Maintenance Steps

Scheduling maintenance steps is one of the activities that the management must undertake in order to maintain the system efficiently and consistently. In the Model of Software Maintenance, maintenance step scheduling serves as the authorization to execute a step. For the steps with a generic primary input, it results in a generic-to-specific input resolution.

Scheduling of the maintenance steps must take into account the current state of the maintenance process, all of the requests for change that are cleared for implementation by the SCCB, the urgency of the required changes to the system, and the imposed sequencing constraints. It also must assure that the consistency of the system (as it is defined in the Model) will be preserved. Since scheduling is based on the changing state of the system configuration, the scheduling decisions depend on time.

The necessary constraints for the correct maintenance step scheduling in the context of the Model of Software Maintenance can be met by the following guidelines:

(3-1) Maintenance task scheduling can be performed only on pending maintenance steps.

(3-2) A necessary condition for a maintenance step to be scheduled is that its primary input is available (see rule (2-24)).

(3-3) A step M_p can be scheduled for implementation iff $\neg\exists$ a step M_q | M_q is implementing or pending and M_q precedes M_p.

(3-4) If step M_p precedes step M_q, then the step M_p will be scheduled before the step M_q.

(3-5) If $\neg\exists$ a "precedes" relationship between steps M_p and M_q, and step M_q precedes another step M_r that has a higher priority than M_p, then the step M_q will be scheduled before the step M_p.

(3-6) If rules (3-4) and (3-5) do not apply, then the step with the higher priority value will be scheduled first.

It should be noted that by the definition of the specific input designation, specific inputs are always available for the maintenance steps. Therefore rule (3-2) influences only pending maintenance steps with a generic primary input designation. If there are several maintenance steps with the same generic primary input, then, since scheduling one of these maintenance steps "ties up" the primary input component (i.e., makes it unavailable to others), none of the other maintenance steps with the same generic input can be scheduled at this time.

Decisions about which of the maintenance steps with the same generic primary input should be scheduled before others are based on rules (3-4)-(3-6). For cases in which a

decision based solely on these rules cannot be reached, the decision process can be augmented using a "rule of thumb" type of scheduling with the oldest pending step first, etc.

A scheduling algorithm that is based upon the Model of Software Maintenance and that satisfies the above constraints is presented in Figure 3-2. This algorithm utilizes graph theory, and it determines vertex basis of an acyclic digraph that represents the relevant relationships between maintenance steps. Since the vertex basis (i.e., a minimal set of vertices that reach all vertices of a digraph) of an acyclic directed graph is unique [Ref. 19], maintenance steps that belong to the vertex basis can be scheduled concurrently without mutual interference or input deadlocks.

In the first phase of the algorithm, a digraph D is created using all existing atomic pending maintenance steps as its vertices. The "precedes" relations between the pending steps are represented as arcs of the digraph D, and (in the second phase) vertex priority values are computed for each vertex by assigning it the maximum of priority values of all vertices that it can reach.

In order to represent the existing "precedes" relations between implementing and pending steps, a dummy vertex M_* is introduced (in the third phase of the algorithm), and the relevant relations are mapped as arcs between the dummy and the pending steps. Assuming that the vertex priority values are non-negative integers with zero representing the lowest priority value, in order to prevent the need to recompute vertex priority values the M_* vertex is assigned priority zero.

Phase four of the algorithm performs generic-to-specific input resolution. In order to do so, the digraph D is partitioned into generic subdigraphs D_i in such a way that each subdigraph contains only those maintenance steps that have the same generic primary input component. It should be noted that in this partition $\cup D_i \subseteq D$ and $D_i \cap D_j = \varnothing$ for $i \neq j$. For each subdigraph D_i, a single maintenance step that is to be scheduled before all other steps in the subgraph is determined, using the vertex priority values that were computed earlier and

1. *Create digraph **D** as follows:*

   a. $\forall M_q \in \mathbf{M}$ *pending atomic step with available primary input* $<=> M_q \in V(\mathbf{D})$.
   b. $\forall M_q, M_p \in \mathbf{D}$, *step* M_q *precedes step* $M_p <=>$ *arc* $(M_q, M_p) \in A(\mathbf{D})$.

2. *Compute vertex priority values:*

   a. *Assign to each vertex* $M_q \in \mathbf{D}$ *a priority value of a step* M_q.
   b. $\forall M_q \in \mathbf{D}$, *assign vertex priority value of* M_q *to the vertex highest priority value of set* $\{M_p \in \mathbf{D} \mid \exists$ *path from* M_q *to* $M_p\}$.

3. *Map "precedes" relation between implementing & pending steps:*

   a. *Add dummy vertex* M_* *to* **D** *with priority value 0.*
   b. $\forall M_q \in \mathbf{D}, ((\exists$ *implementing* $M_p \in \mathbf{M} / M_p$ *precedes* $M_q) => arc\ (M_*, M_q) \in A(\mathbf{D}))$.

4. *Perform generic-to-specific input resolution:*

   a. *Partition the digraph **D** into subdigraphs* $\mathbf{D}_i$ *such that all vertices in each subdigraph have the same generic primary input.*
   b. $\forall \mathbf{D}_i \in \mathbf{D}$ *determine step* $M_q \in \mathbf{D}_i$ *that should be scheduled before all other steps* $M_p \in \mathbf{D}_i$, *using computed vertex priority values and rules (3-2)-(3-6).*
   c. $\forall M_p \in \mathbf{D}_i$, $p \neq q$ *add arcs* (M_q, M_p) *to* **D**.

5. *Check conflicts with inducing steps:*

   a. *Find the vertex basis **B** of digraph **D**.*
   b. $\forall M_q \in \mathbf{B}$ *with primary input* C_j, *compute* C_{Cj}. *If* $C_{Cj} \neq \emptyset$, *then* $\forall M_p \in \mathbf{B}$, *with primary input* $C_i \in C_{Cj}$ *add arcs to digraph **D** as follows:*

      1) *If priority of step* $M_p >$ *priority of step* M_q, *then add arc* (M_p, M_q).
      2) *If priority of step* $M_p \leq$ *priority of step* M_q, *then add arc* (M_q, M_p).

6. *Compute scheduling step sequence:*

   a. *Find the new vertex basis **B'** of digraph **D** and remove the dummy vertex* M_* *from it.*
   b. *Sort **B'** in descending order of the computed vertex priority values.*

7. *Expand inducing steps:*

   a. $\forall M_q \in \mathbf{B'}$ *with primary input* C_j, *if* $C_{Cj} \neq \emptyset$, *then* $\forall C_i \in C_{Cj}$ *create induced steps* $M_{q(n)}$ *with primary input* C_i *and priority value of vertex* M_q *as it was computed in phase 2.*
   b. $\forall$ *inducing steps* $M_q \in \mathbf{B'}$ *place all* $M_{q(n)}$ *immediately following the step* M_q.

Figure 3-2: Maintenance Step Scheduling Algorithm.

applying rules (3-2)-(3-6). These rules should be applied with respect to the whole digraph D, i.e., they should not be limited to subgraph D_i while considering some maintenance step as a candidate. If an automatic decision cannot be reached (especially due to the non-uniqueness of a generic input designation in the case of parallel genealogies), manual intervention may be required. Once the proper maintenance step is determined, arcs between the appropriate vertex and other vertices in the subdigraph D_i are added to digraph D. Since the chosen vertex in each D_i has no incoming arcs, the digraph D remains acyclic.

At this point, the digraph D represents all atomic maintenance steps and all explicit and implicit precedence relationships between them. The vertex basis B of digraph D consists of all pending maintenance steps that are not preceded by some other steps, and, therefore can be scheduled for execution at this point in time. Since the digraph D is acyclic, its vertex basis is unique, i.e., there are no other maintenance steps that can be scheduled at this point in time since they do not belong to the vertex basis of digraph D.

If an inducing maintenance step belongs to the vertex basis of digraph D, then a conflict may arise between this inducing step and some other maintenance steps which are also in the vertex basis and that have a primary input component which is influenced by the inducing step. In order to eliminate such conflicts, the fifth phase of the algorithm constructs additional dependency arcs in D between steps in B. The direction of these arcs is determined by comparing the computed priority values of the conflicting steps - from the vertex with a higher priority value towards the vertex with the lower priority value. Since the vertices in B do not reach one another, and due to the transitivity of the "less-then-or-equal" relationship between priority values, the resulting digraph D is still acyclic.

In the sixth phase a new vertex basis B' (which is also unique due to the acyclic nature of D) is computed. The dummy vertex M. is removed from B', and its maintenance steps are sorted in descending order of the computed vertex priority values.

Phase seven is the last phase of the algorithm in which the inducing steps in B' are expanded, i.e., necessary induced steps are created. The induced steps are assigned the priority value of their inducing step and are placed directly following it in the sorted step sequence. The resulting sequence of maintenance steps represents all maintenance steps sorted in the decreasing order of their computed priorities. These can be executed simultaneously without compromising system consistency, provided that necessary synchronization concerning induced and inducing maintenance steps is done. This sorted sequence is to be used for the execution of maintenance steps, for the programmer's job assignments, and planning.

Since the conflicts between maintenance tasks that may hamper their execution are resolved at scheduling time, the execution of the scheduled maintenance steps is fairly simple. The main phases of maintenance step execution are shown in Figure 3-3. Note that in cases with a specific primary input the first phase of the algorithm is redundant. The third phase of the maintenance step execution algorithm requires determination of non-primary inputs of the maintenance step. An algorithm to determine the input set of a scheduled maintenance step is shown in Figure 3-4.

In order to compute the input set of a maintenance step for non-primitive primary input the algorithm is propagating non-primary inputs from a maintenance step that created the primary input component. In case of primitive primary input, the "is-component-of" relationship is used to determine the non-primary inputs. The third phase of the algorithm enforces system consistency in case of induced step input set determination. It should be noted that in such cases while the direct descendent of C_k has not been completed yet, there already exists a "placeholder" for it in the configuration graph G. Since $C_k \in C_{Cj}$, it is used as a primary input by some other induced step that is triggered by a change in the component C_j. The contents of the direct descendent of component C_k will be produced when all induced steps triggered by C_j are completed (virtually simultaneously, due to the semi-atomic behavior of induced steps).

1. *If primary input C of the step M_q is generic, then take the latest descendent of C as a specific input.*

2. *If the step M_q is induced by some other step M_p, then wait until step M_p starts implementation.*

3. *Compute the non-primary inputs of M_q.*

4. *Start implementation of the maintenance step M_q.*

5. *if the step M_q is not induced by any other maintenance step and it is itself an inducing step (i.e. set $C_C \neq \emptyset$), then wait until all induced steps $M_{q(n)}$ are completed.*

6. *Complete M_q.*

Figure 3-3: Maintenance Step Execution Phases.

As mentioned earlier, there may be a need for manual intervention in order to resolve ambiguous primary input designations and to make sure that all non-primary inputs are accounted for. Phase four of the algorithm in Figure 3-4 explicitly allows for such interventions whenever necessary.

2. Maintenance Job Assignment

Assigning maintenance jobs (steps) to a programmer is one of the basic tools for planning and control of the maintenance process. The job assignment process deals with matching scheduled maintenance steps to available maintenance programmers. Effective job assignment planning requires access to information about the programmer work force, thus allowing integration of some of the organizational aspects of the maintenance team within the Model of Software Maintenance analogous to coupling of technical aspects of software productions with human resource aspects as formulated by Abdel-Hamid and Madnick [Ref. 20].

We can view the available human resources (i.e., the maintenance programmer work force) as a set **H** that consists of entities $H_i \in \mathbf{H}$ where each H_i represents a programmer that is a full or part time member of the maintenance team, i.e., $\mathbf{H}=\{H_i \mid H_i$ represents a maintenance programmer$\}$.

1. *Set $I_{M_q} = \{C_i \mid C_i$ is primary input of $M_q\}$.*

2. *Create a temporary set of components S as follows:*

   a. *if $C_i \in P$ (i.e., C_i is primitive) then $S = \{C_k \mid C_k$ is a component of $C_j\}$.*
   b. *Otherwise $S = I_{M_p}$ such that $C_i \in O_{M_p}$.*

3. *$\forall\ C_k \in S$ perform the following:*

   a. *if M_q is an induced step triggered by a change in C_j and $C_k \in C_{C_j} \cup \{C_j\}$, then add the next direct descendent of C_k to I_{M_q}.*
   b. *Otherwise add C_k to I_{M_q}.*

4. *Verify that $I_{M_q(n)}$ contains all the necessary components, update manually if necessary.*

Figure 3-4: Determining Input Set of a Maintenance Step M_q.

For each entity $H_i \in H$ necessary minimal attributes can be defined as follows:

- Potential workload per unit of time (e.g., month).
- Actual workload per unit of time.
- Subject/System/Language expertise.

These attributes will be used later by the job assignment algorithm. Additional attributes for maintenance programmers (e.g., expertise levels) may be defined as well and they may be used in refinements of this algorithm or for additional purposes.

A general algorithm for assigning currently scheduled maintenance steps to programmers is presented in Figure 3-5 . The algorithm begins by scheduling pending maintenance steps, and then for each scheduled step that has not been assigned yet it determines the programmers that have the expertise to perform the required maintenance activity. If a programmer with the required expertise and a free potential workload is found, the algorithm assigns the maintenance step to him and updates the programmer workload. This algorithm does not guarantee a solution optimized with respect to use of the available programmer work force or any other criteria, but it seems that it can be modified to do so.

1. *Compute set M_{sh} = {M_q / M_q is currently scheduled but not yet assigned}.*
2. *∀ scheduled steps $M_q \in M_{sh}$ perform the following:*
   a. *Determine set H_{M_q} = {H_i / $H_i \in H$ has the expertise to do M_q}.*
   b. *If ∃ $H_i \in H_{M_q}$ with free workload potential then*
      1) *Assign M_q to H_i.*
      2) *Add estimated effort to execute M_q to the actual workload of H_i.*
   c. *Otherwise can't plan on executing M_q at this time.*

Figure 3-5: Job Assignment Algorithm.

The job assignment algorithm takes into account the following restrictions:

- Not every programmer can perform every maintenance job.
- Workload potential of the maintenance programmer is finite.

It should be noted that the job assignment algorithm explicitly shows that due to the previous constraints, there may exist some scheduled maintenance tasks that cannot be performed at a given point in time. Such cases should be addressed by the management of the maintenance team/organization, possibly outside the scope of the Model of Software Maintenance.

Note that although the algorithm presented in Figure 3-5 does not explicitly mentions possibility of assignment of the same maintenance step to more than one programmer, it can be modified accordingly to support such feature.

Assigning a maintenance steps to programmers is equivalent to defining a relationships between two disjoint non-empty sets: a set of the currently scheduled maintenance steps (i.e., the set {M_q | M_q is currently scheduled}) and the set **H** that represents available human resources. These relationships define a bipartite digraph, and therefore the job assignment becomes a matching problem. Such problems have been well studied, and algorithms for finding optimal solution according to some defined criteria are known (e.g., the

"Hungarian method") [Ref. 21]. Use of such algorithms on top of the job assignment algorithm shown in Figure 3-5 can provide solutions that may maximize the number of assigned maintenance tasks, minimize the total time to complete all scheduled maintenance tasks, optimize work force utilization, etc.

3. Maintenance Process Simulation

The job assignment algorithm presented in the previous section is based upon the set of currently scheduled maintenance steps and provides planning for a particular unit of time (e.g., month). In cases when the set of currently scheduled maintenance steps is too small to complete the planning for this unit of time (i.e., to provide scheduled jobs for all available programmers for the whole unit of time), the need for simulation of the maintenance process arises. Such simulation is useful for other needs as well, e.g., completion forecast of a particular maintenance task, analysis for the maintenance team size, future trends planning, etc.

A general algorithm for simulation of the maintenance process is presented in Figure 3-6. This algorithm is based upon presuming completion of a maintenance step with the smallest execution time span. This is coupled with the iteration of the pending maintenance step scheduling and the programmer job assignment until the simulation goals are met. The scheduling and job assignment cycles of the simulation use algorithms presented earlier in this chapter.

The accuracy of the maintenance process simulation depends among other factors on the following assumptions:

- Execution time estimates for all maintenance steps are correct.
- Maintenance steps are not aborted during the simulation.
- The work force pool remains static during the simulation.
- There are no new maintenance steps introduced during simulation.

While simulation limits have not been reached perform:

a. Assign jobs to programmers.

b. Find all assigned maintenance steps with minimal estimated execution time.

c. Simulate completion of all maintenance steps found in previous phase.

d. Re-assign (and re-schedule) maintenance tasks.

Figure 3-6: Maintenance process simulation.

The last two of the previous assumptions may be changed in order to achieve better simulation or perform sensitivity analysis. For example, we can define a new hypothetical maintenance team and check how hiring/firing of programmers may affect the maintenance process. We also can simulate influx of new pending maintenance steps (by using some predetermined scenario that may be based upon the maintenance history of the system) in order to achieve a higher degree of reality in the simulation of the maintenance process.

Additional improvement of simulation precision can be achieved by incorporating new programmer training process factors into the simulation model. For example, using the ideas and notation of [Ref. 22] and [Ref. 23], we can define for a new programmer an "Average Assimilation Delay" at a granularity level of the amount of the workload potential per unit of time (e.g., man-hours per month) which is going to be spent on learning and "social orientation". We can introduce an effective workload function $W(H_i,t)$ that will compute for a programmer $H_i \in H$ the effective workload potential (i.e., workload potential without the effort spent for learning) as it depends on time of employment t. Also, we can introduce an additional function $W^*(H_j,t)$ that will provide the effective workload potential for an experienced programmer $H_j \in H$ that is tutoring a new programmer (e.g., H_i). In analogy, additional workload potential functions can be introduced that take into account fluctuations of a workload potential for the work force depending on time of the year, holidays, etc. Naturally, all of the effective

workload potential functions mentioned above are dependent on organizational policies, structures and experiences.

Incorporating both the programmer's expertise data and the effective workload functions into the maintenance process simulation will enable a thorough analysis of the dynamics of maintenance team staffing. Coupled with other staffing decision factors described by Abdel-Hamid in [Ref. 22], this analysis may help in reaching correct decisions about the a staffing policy for the maintenance team.

The ability to simulate the future of the maintenance process based upon the current state of the maintenance process is a very important capability for the management of the maintenance team. It may provide a sound basis for a variety of decision support tools targeted to improve the efficiency and cost-effectiveness of large systems' maintenance.

4. Controlling the Maintenance Process

In essence, controlling the software maintenance process does not differ significantly from controlling a software project. The issue of control of software development and maintenance is to minimize surprises along the way (see [Ref. 24]). For the purpose of simplicity, we will view the control of the maintenance process as checking progress against plans and flagging possible crisis situations.

The Model of Software Maintenance defines structure which allows a high degree of control over the maintenance process. This is mainly due to the tight integration between the configurational and managerial aspects of the maintenance process management [Ref. 18]. In the context of an engineering database this integration can provide automated support for the control of the maintenance process. This support allows it to provide the required services that were mentioned in the beginning of the chapter, e.g., to provide maintenance process status and to track maintenance progress.

Previously in this chapter principles and algorithms for planning of the maintenance process were described. Since all the necessary data about the operational planning of the maintenance process is contained within the maintenance-oriented engineering database, and since the Model of Software Maintenance defines distinct execution states for each maintenance step and all possible relationships between the maintenance steps, we can see that the necessary information about the current state of all maintenance steps is stored in the engineering database as well. This information is automatically updated conforming to the ongoing changes as the result of the execution of maintenance tasks. Software configuration management facilities of the engineering database discussed in the next section of this chapter provide an additional level of direct control over the maintenance process. For example, start of a maintenance step implementation or completion of a maintenance task are automatically recorded in the engineering database, and any deviation in the dates of these events may be detected immediately (by checking the relevant dates against the plans).

As a result, various control reports about the ongoing state of the maintenance process can be generated utilizing the information in the maintenance-oriented engineering database. Some examples of such reports are listed below:

- Current state of the maintenance tasks, i.e., what maintenance steps are awaiting decisions, what steps are in the pending state, how many maintenance steps are currently under execution, what maintenance tasks have been completed in a given unit of time, etc.
- Summary reports about the completed maintenance tasks with all relevant information (e,g., estimated and actual programmer resources).
- Exception reports, e.g., what maintenance steps where to be completed for a given deadline but are not.

Assuming a programmer effort report facility within the engineering database environment that records efforts the programmers spent for each maintenance step they performed, additional control over the maintenance process can be provided by allowing compilation of reports like the following:

- Over-expenditure of maintenance effort, i.e., in which maintenance tasks the effort spent exceeds planned effort.
- Late start of maintenance task execution, i.e., what maintenance steps should have started the implementation phase but have not done so.

The required reports can be produced automatically (triggered by some event or criterion) or periodically (e.g., weekly or monthly), and since the most of the information about the progress of the maintenance process is gathered automatically, the effort required to produce the necessary reports is relatively small. Also, the required reports that reflect the current state of the maintenance process are highly available, i.e., they may be produced at any time.

B. CONFIGURATION MANAGEMENT AND CONTROL

The importance of proper configuration and control procedures for the success of the software maintenance was described earlier in Chapter I. According to the ANSI/IEEE Guide to Software Configuration Management [Ref. 11], software configuration management provides a common point of integration for planning, controlling and implementation activities of a software project during its lifecycle, is practiced within the management context of the software project, and provides the maintenance management with the visibility of the maintenance process and means to control it.

The Model of Software Maintenance described in Chapter II was developed with the recognition of the importance of software configuration management issues. It uses system software configuration as one of the bases of the model and it supports concepts required to provide the necessary configuration management and control tools for a maintenance-oriented engineering database.

The Model of Software Maintenance supports basic features required for any successful maintenance configuration management and control system (see [Ref. 17] and [Ref. 25]):

- Record of system evolution history is provided automatically since all components and maintenance steps with their input and output sets are retained within the engineering database.

- "Forking" of a component evolution line into two or more independent evolution lines/trees is supported by the Model.
- Support of product release is provided since the Model captures all the necessary relationships between the components, their grouping into systems/products, and it defines and imposes consistency as well.
- Version tracking is supported as part of the system evolution record.
- Gathering of the necessary information for management activities (e.g., maintenance planning, progress tracking) is provided since all the required data is defined within the Model framework.
- Support of different products within the same configuration management environment is provided directly by the Model together with component reusability within scope of different systems/products.

As previously defined, components of the Model of Software Maintenance are non-re-derivable components, implicitly assumed to be source code modules. But besides the source code modules, the Model can also support additional types of software modules (also are non-re-derivable in terms of [Ref. 16]) as its components, e.g.,:

- Specifications.
- User documentation.
- Programmer documentation.
- Test data and test generation procedures.
- Data dictionaries.

Inclusion of these types of software modules within the framework of the Model of Software Maintenance expands the notions of both the configuration management and the control to all components and related software modules that are manipulated during the maintenance process. It is implicitly supported by the Model's system/product definition mechanisms. Proper manipulation of these new types of software components may be further augmented by introduction of additional relationships.

Consider a ***depends-on*** relationship between a system component and its documentation. Such relationship can be used to trigger documentation updates as a result of changes in its

related component, thus enforcing consistency in the documentation domain of the maintained software system. The "depends-on" relationship can be used to tie components with their specifications, test data, test generation procedures, etc. Also, reverse links can be defined, e.g., a "depends-on" relationship between the specifications and the components that implement them. Although the "depends-on" relationship is not defined within the Model of Software Maintenance, it can be implemented within the framework of an engineering database based upon the Model and tailored to the specifics of the target environment.

Since the Model of Software Maintenance was built upon to comply with the ANSI/IEEE Guide to Software Configuration Management [Ref. 11], it supports the following:

- Baseline concept.
- Software product promotions/releases.
- Software product versions/revisions.
- Software product structure.
- Software libraries.
- Software product generation.

In the Model of Software Maintenance a ***baseline*** B of a system S (denoted as B^S) is a set $\{C_i\}$ of unique representatives of the genealogy trees that comprise the system S, i.e.,:

(3-7) $B^S = \{C_i | C_i \in T_k \text{ and } T_k \in S\}$ and $\forall\ C_i, C_j \in B^S\ C_i, C_j \in T_k \Rightarrow C_i = C_j$.

A baseline of a system must be complete and consistent, i.e., the following holds:

(3-8) $\forall\ C_i \in B^S$, if $C_i \in P$, then $\forall\ C_j$ such that C_j is a component of C_i, $C_j \in B^S$. Otherwise $\exists\ M_q$ such that $C_i \in O_{Mq}$, and $\forall\ C_j$ non-primary inputs of M_q, $C_j \in B^S$.

A baseline is unique in the context of a system, i.e.,:

(3-9) $\forall\ B^S_k, B^S_n$ of a system S, if $\forall\ C_i \in B^S_k\ \exists\ C_j \in B^S_n$ such that $C_i = C_j$ and $|B^S_k| = |B^S_n|$, then $B^S_k = B^S_n$.

The baseline concept is important for configuration management and control. It introduces the dynamic aspects of system evolution, provides foundation for change management, and serves as a basis for coordinating engineering activities concerned with software evolution.

The evolution of a software system can be viewed as an ordered set of its baselines, where the ordering is based upon the time of a managerial decision to introduce (or roll-back to) a baseline (and not on the time of baseline creation, since it does not take into account the possibility of roll-backs to previously defined baseline). It should be noted that the baseline ordering is not directly related to the component ordering of the digraph G which is the basic structure of the Model of Software Maintenance, i.e., there is no direct relationship between topological ordering of components, and the ordering of baselines these components belong to.

The software product promotion/release concept is supported by enforcing the formality of the managerial decisions concerned with the changes made to the baselines of the software product. In plain words, definition of a baseline (by defining components that belong to it) requires an explicit managerial decision, part of which is to decide whether the new baseline is to be released to the user or not.

Use of the baseline concept allows implicit support of the versions/revisions concept. From the moment the new baseline is identified, all changes between the new baseline and the previous baselines are identified as well, and any new functional capabilities added to the system (leading to a new version) can be identified and appropriately documented.

The software product structure is supported by a system/product concept of the Model of Software Maintenance which also supports the hierarchical decomposition of software products by the use of intermediate subsystems. For example, a software configuration item (CI in terms of the IEEE Guide to Configuration Management) can be defined as a system S that consists of subsystems S_i (i.e., $S=\cup S_i$ and $S_i \cap S_j=\varnothing$ for $i \neq j$) where each subsystem is a set of genealogy trees (i.e., $S_i = \{T_k\}$). Such representation defines a three level product hierarchy, and it can be expanded to include more levels of hierarchy by defining intermediate subsystems as required.

The concept of multiple software libraries (i.e., Controlled, Dynamic and Static libraries of the IEEE Guide to Configuration Management) is implicitly supported by the Model of

Software Maintenance. The main configuration repository of the Model (i.e., the digraph G defined in Chapter II) corresponds directly to the master (Controlled) library, and the Model of Software Maintenance centers on its structure and operations. The Dynamic library in which the hands-on software maintenance/development is performed and the Static library that holds the released versions and revisions of the software product are outside of the direct scope of the Model. These libraries can (and should) be defined by the configuration management according to organizational policies and practices. They should not influence the structure and operation of the master library except for trivial interfacing between them.

The Model of Software Maintenance supports the following operations on the managed components within the master (Controlled) library:

- Checkouts of software components to programmers according to their requests.
- Authorized check-in from programmer as a result of completion of the implementation phase (i.e., when the maintenance step makes the transition from implementing to completed state).
- Baseline definition and baseline rollback as a result of an explicit managerial decisions.
- Status accounting and history reports.

It should be noted that since there are no locks on the entities of the master library, and all the changes to it must be authorized in advance by a managerial decision within the consistency constrains imposed by the Model, there are no concurrent update conflicts. Thus the READ ONLY access to components within the master library is always supported as illustrated above by the absence of authorization for component checkout.

Although the Model of Software Maintenance deals only with the non-re-derivable components, it implicitly supports the concept of software product generation. The Model's "is-component-of" relationships, system/product structure definition and the consistency enforced by the Model allow automatic generation of software products. Since the actual generation of the software product requires additional information that is application dependent and concerns both the target machine (e.g., machine type, memory size, etc.) and the product specifics, it is

outside the scope of a general model (like the Model of Software Maintenance), and it should be implemented within the scope of the configuration management system. Since the maintained components may be of different types (e.g., written in different programming languages), different generation procedures may be required. In a such case, a sophisticated product generating sub-system (that should be a part of the configuration management and control system) could be implemented. Such a subsystem may utilize the configuration knowledge base ideas described in [Ref. 16], and its capabilities can be augmented by a rule-based expert subsystem that utilizes the component's attributes (e.g., component's programming language type can be used to determine the appropriate compiler to be used), and has additional advanced features (e.g., caching of intermediate compiled software product components for overall efficiency of the product generation). Naturally, since the Model of Software Maintenance enforces baseline consistency of software products, the resulting generated software product is consistent with itself (i.e., within the scope of its constructing components), and it will reflect the authorized maintenance work.

Since the Model of Software Maintenance deals with all facets of the maintenance process information flow (from user request for a change, throughout SCCB deliberations until the maintenance step implementation and completion), the model actually supports the system configuration management and control actions throughout the entire process of software maintenance. The model also combines into one coherent framework the information and the control of the configuration management and control functions. It allows automation of the mechanics of the pure configuration management and control process (e.g., change initialization, status accounting, change control and tracking, etc.), and on-line tools to observe and control the dynamics of software maintenance process as a whole.

A configuration management and control system based upon the Model of Software Maintenance has additional aspects: due to the discipline imposed by the model it supports improved communication and coordination with the user (e.g., throughout well defined

procedures for change initiation), and it improves visibility of priorities of the requested changes and corresponding decisions about maintenance step implementation. As a result, software problem reports and change initiations are promptly and properly recorded, the implementation analysis and decision phase is enforced for all maintenance activities, and planning and control functions become part of the maintenance process. Also, since all changes to the system's configuration must be approved by appropriate maintenance management levels, all concurrency conflicts (e.g., concurrent update of the same system component) are solved at these levels and they do not filter down to the programmer work force. Another beneficial aspect of a configuration management and control system based upon the Model of Software Maintenance is to help improve the Quality Assurance (QA) of the maintenance process by explicitly enforcing consistency (e.g., baseline consistency) and implicitly enforcing checks and formal decisions associated with the QA activity.

C. GATHERING AND PROCESSING METRICS AND STATISTICS

One of the important aspects of the maintenance-oriented engineering database is the maintenance process information gathering and processing. This information is very helpful in evaluating the effectiveness of the maintenance process and/or maintenance techniques as defined in [Ref. 9]. Also, it allows identification of error-prone components and a more accurate prediction of future maintenance costs than is usually available [Ref. 26].

Since the Model of Software Maintenance views system components as immutable objects (see Chapter I) and deletion of these components is prohibited, it is obvious that the Model retains historic information. Therefore, a maintenance-oriented engineering database built upon the Model of Software Maintenance can serve as a historic database of the system evolution. Because both the system components and the maintenance steps are elements of the Model of Software Maintenance, the information in this historic database pertains to aspects of maintenance process history (i.e., system evolution) and to component contents.

Since software faults are treated as change requests, and the engineering database contains the information about the faults themselves and all maintenance steps that were performed in order to eliminate them, we can see that the maintenance-oriented engineering database contains the necessary quality accounting information as well.

Processing the data contained in the maintenance-oriented engineering database can provide the necessary information to evaluate the performance of the maintenance process for the organization as a whole or for a particular maintenance team, system, etc. Besides the basic maintenance summary reports defined by Swanson in [Ref. 9], additional computations and summary reports can be produced utilizing the data in the engineering database, e.g.,:

- Average number of lines of code (LOC) changed per maintenance task for system as a whole or for particular subsystems.
- Average programming effort required for the execution of a maintenance task for the whole system or for particular subsystems.
- Maintenance programmer productivity for the whole system or as function of a particular component, subsystem or programming technology (e.g., computer language).
- *Programmer fault insertion rates* as a function of time, subsystem, component or programming technology (e.g., computer language).
- Annual rate of user change requests as a function of time or subsystem.
- Annual Change Traffic (ACT) values for the whole system and/or the system components.
- Maintenance effort distribution as a function of subsystems or system components.
- System/subsystem development spoilage as function of time.
- Changes in system complexity (e.g., McCabe cyclomatic measure) as a function of system evolution.

As shown above, the required computations and reports can be produced for the whole system, for particular subsystems and components, or for a particular programming technology (e.g., computer languages and tools). This feature may be very useful for systems that consist of components that employ different programming technologies and techniques, which is a very likely feature in a large software system.

The analysis that can be performed using the historic information in the engineering database can be of great help in high level prediction/planning of the maintenance process. For example, we can estimate the financial impact of system maintenance utilizing COCOMO [Ref. 8] and using correct ACT values that were computed from the information in the engineering database. In general, maintenance process historical information can help management to predict more efficiently the maintenance costs. As a result, it may also help to reduce the cost of system maintenance - two important goals of maintenance team/organization management.

Most of the information required for the maintenance metrics and statistic is collected/recorded automatically during the execution of a maintenance process and already exists in the engineering database. Therefore, unlike current practices of metrics and statistics data gathering that may consume a considerable effort [Ref. 24], there is a very low overhead involved in retrieving the necessary data for the required metrics and statistics computations. Also, the availability of the necessary data allows ad hoc generation of statistical reports that reflect the current state of the maintenance process.

IV. CONCLUSION

This thesis presents a Model of Software Maintenance for large scale computer systems. In the first chapter of this work the issues of software maintenance were examined with emphasis on the specific maintenance problems of military software, and the need for a maintenance-oriented engineering database is established. Also a need for a consistent underlying model for the engineering database is identified. Such a model, called the Model of Software Maintenance, is proposed in the second chapter of the thesis. The third chapter shows how to utilize the proposed Model for the purposes of maintenance process management, assuming that the Model serves as a theoretical basis for a maintenance-oriented engineering database. In particular, algorithms are provided for maintenance task scheduling, programmers job assignment, maintenance process planning, etc. Also, benefits of integration and automation of different aspects of the maintenance process (e.g., gathering and processing maintenance metrics and statistics) are shown.

Special emphasis was given in the development of the Model of Software Maintenance to comply with the existing standards, e.g., the Guidance on Software Maintenance document from the USA National Bureau of Standards and the ANSI/IEEE Guide to Software Configuration Management were consulted and followed as closely as possible throughout the Model definition.

Also, special attention was given in definition of the Model of Software Maintenance to stay within the theoretical aspects of mathematical modelling without incorporating specific organizational and implementation aspects into the Model. This was done to keep the model abstract enough to be used as a basis for engineering database implementation in different organizations, each with its own organizational structure and policies.

The Model of Software Maintenance was developed using an integrated approach to software maintenance. Change control, configuration management and control, relationships between software components, managerial aspects of controlling the maintenance process and managing available human resources are consolidated into one framework. This feature of the Model is unique, i.e., we do not know about any other model of software maintenance or production that does so. The existing practical approaches to change control and configuration management (i.e., implementations of change control and CM systems) address single aspects of the software maintenance and provide little integration capabilities with other facets of maintenance process.

A. RESEARCH CONTRIBUTIONS

The main contribution of this research is the development of the Model of Software Maintenance.

The Model of Software Maintenance provides a unique, integrative framework that describes the dynamic, configurational and managerial aspects of software maintenance process. It imposes a well defined structure on the maintenance process, provides control over the process lifecycle, allows automatic gathering of maintenance statistics, preserves maintenance history, and allows rollback of the "ill fated" maintenance tasks influence.

The Model of Software Maintenance provides a mathematically sound basis for implementation of a maintenance-oriented engineering database, and it allows derivation of algorithms required for effective and efficient utilization of such database. Some of these algorithms were introduced in Chapter III of this thesis, some are still to be developed in the future. The resulting engineering database is capable of supporting the required information flow and control necessary for successful management of the maintenance process.

The Model of Software Maintenance also imposes a well defined discipline on the process of software maintenance within the scope of the organization that uses the software

system, and, as a result, it contributes to improved communication between users and the system maintainers.

B. FUTURE DIRECTIONS

It appears that at this stage of its development the Model of Software Maintenance is complete and consistent by itself.

It seems logical to advise that future theoretical efforts within the framework of the Model should concentrate on improvement of the existing algorithms and development of nLatabase, e.g., access to the data, processing of the raw data using specific algorithms and processing data requests for the database users (e.g., SCCB members, maintainers, managers, etc.).

Since all of the entities in the Model of Software Maintenance can be represented as data entities, it seems that the heart of the functional database kernel can be implemented using a regular database[5] system (e.g., currently available commercial DBMS). The functionality of the kernel should be provided by additional software that is built upon the DBMS services. This software should implement the necessary algorithms (e.g., maintenance step scheduling, job assignment, etc.) traversing the mathematical structures of the Model of Software Maintenance.

The flexible engineering database kernel interfaces should be responsible for the task of adapting the kernel to specific organizations. Such adaptation depends on the details of the organizational structure and peculiarities of the maintenance process in each organization. Since these specifics are difficult to foresee, the kernel interfaces must be flexible and sufficiently general. Naturally, a choice of database technology for the functional kernel (e.g., relational or object oriented) should not impede the flexibility and generality of the kernel interfaces.

---

[5] Regular here means usual information database and is in contrast to the maintenance oriented engineering one.

The implementation of the maintenance-oriented engineering database based upon the Model of Software Maintenance should address issues of the database schema design, appropriate database technology choice, the database operation and evaluation.

Although the Model of Software Maintenance is oriented (as its name suggests) towards the maintenance phase of the software system life cycle, it can also be adapted to software development, especially when an evolutionary software development approach is used. Because this thesis concentrates on the maintenance aspects, no attempt was made to pursue this issue within the scope of this research. Doing so in the future may be a good idea, especially after the Model proves itself in a successful implementation.

LIST OF REFERENCES

1. Arthur L.J., *Software Evolution - The Software Maintenance Challenge*, John Willy & Sons, Inc, 1988.

2. Martin R.J., and Osborne W.M., *Guidance on Software Maintenance*. National Bureau of Standards, U.S. Departments of Commerce, December 1983.

3. Schneidwind N.F., *The State of Software Maintenance*, IEEE Transactions on Software Engineering, Vol. SE-13 No. 3. March 1987.

4. Lehman, M.M. and Belady L.A., *Program Evolution, Process of Software Change*, ACADEMIC PRESS 1985.

5. Lientz B.P., *Issues in Software Maintenance*, ACM Computing Surveys, Vol.15, No 3, September 1983. pp. 171-178.

6. Day R., *A History of Software Maintenance for a Complex U.S. Army Battlefield Automated System*, IEEE Proceedings of Conference on Software Maintenance. IEEE, November 1985. pp. 181-187.

7. Lientz B.P. and Swanson E.B., *Software Maintenance Management*, ADDISON-WESLEY 1980. pp. 81.

8. Boehm B.W., *Software Engineering Economics*, PRENTICE-HALL 1981.

9. Swanson E.B., *The Dimensions of Maintenance*, Proceedings of 2nd International Conference on Software Engineering. IEEE. October 1976. pp. 492-497.

10. Lientz B.P. and Swanson E.B., *Problems in Application Software Maintenance*, Communications of the ACM, Vol. 24, no 11, November 1981 pp. 763-769.

11. *IEEE Guide to Software Configuration Management*, ANSI/IEEE Std. 1042-1987, Technical Committee on Software Engineering of the Computer Society of IEEE, 1988.

12. Campbell R.H., and Terwilliger R.B., *The SAGA Approach to Automated Project Management*, Advanced Programming Environments, Proceedings of an International Workshop, Trondheim, Norway,June 1986. SPRINGER-VERLAG pp. 142-155.

13. Berzins V., and Luqi, *Software Engineering with Abstractions: an Integrated Approach to Software Development using ADA*, ADDISON-WESLEY 1989.

14. McClure C.L., *Managing Software Development and Maintenance*, VAN NOSTRAND REINHOLD Co. 1981. pp. 141.

15. Borison E., *A Model of Software Manufacture*, Advanced Programming Environments, Proceedings of an International Workshop, Trondheim, Norway,June 1986. SPRINGER-VERLAG pp. 197-220.

16. Heimbigner D. and Krane S., *A Graph Transform Model for Configuration Management Environments*, Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. 1988. pp. 216-225.

17. Taylor B., *A Database Approach to Configuration Management for Large Projects*, IEEE Proceedings of Conference on Software Maintenance. IEEE, November 1985. pp. 15-23.

18. Branch M.A., Jackson M.C., and Laviolette M.C., *Software Maintenance Management*, IEEE Proceedings of Conference on Software Maintenance. IEEE, November 1985. pp. 62-67.

19. Roberts F.S., *Discrete Mathematical Models*, Prentice-Hall, Inc. 1976. pp. 42-49.

20. Abdel-Hamid T.K. and Madnick S.E., *Lessons Learned from Modeling the Dynamics of Software Development*, Communication of the ACM, Volume 32 Number 12. December 1989. pp. 1429-1455.

21. Dossey J.A., and others, *Discrete Mathematics*, Scott, Foresman and Company, 1987. pp. 233-272.

22. Abdel-Hamid T. K., *The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach*, IEEE Transactions on Software Engineering vol. 15, No 2, February 1989.

23. Abdel-Hamid T.K., *A Study of Staff Turnover, Acquisition, and Assimilation and Their Impact on Software Development Cost and Schedule*, Journal of Management Information Systems. Summer 1989, Vol 6, No. 1. pp. 21-40.

24. DeMarco T., *Controlling Software Projects: Management, Measurement & Estimation*, Yourdon Press, Prentice-Hall, Inc. 1982.

25. Horne C.G. and Seeger R., *An Advanced Configuration Management Tool Set*, IEEE Conference on Software Maintenance. IEEE, 1988. pp. 229 - 234.

26. Schaefer H., *Metrics for Optimal Maintenance Management*, IEEE Proceedings of Conference on Software Maintenance. IEEE, November 1985. pp. 114-119.

BIBLIOGRAPHY

1. Abdel-Hamid T. K., *The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach*, IEEE Transactions on Software Engineering vol. 15, No 2, February 1989.

2. Abdel-Hamid T.K., *A Study of Staff Turnover, Acquisition, and Assimilation and Their Impact on Software Development Cost and Schedule*, Journal of Management Information Systems. Summer 1989, Vol 6, No. 1. pp. 21-40.

3. Abdel-Hamid T.K. and Madnick S.E., *Lessons Learned from Modeling the Dynamics of Software Development*, Communication of the ACM, Volume 32 Number 12. December 1989. pp. 1429-1455.

4. Arthur L.J., *Software Evolution - The Software Maintenance Challenge*, John Willy & Sons,Inc, 1988.

5. Berzins V., and Luqi, *Software Engineering with Abstractions: an Integrated Approach to Software Development using ADA*, ADDISON-WESLEY 1989.

6. Boehm B.W., *Software Engineering Economics*, PRENTICE-HALL 1981.

7. Borison E., *A Model of Software Manufacture*, Advanced Programming Environments, Proceedings of an International Workshop, Trondheim, Norway,June 1986. SPRINGER-VERLAG pp. 197-220.

8. Branch M.A., Jackson M.C., and Laviolette M.C., *Software Maintenance Management*, IEEE Proceedings of Conference on Software Maintenance.IEEE, November 1985. pp. 62-67.

9. Campbell R.H., and Terwilliger R.B., *The SAGA Approach to Automated Project Management*, Advanced Programming Environments, Proceedings of an International Workshop, Trondheim, Norway,June 1986. SPRINGER-VERLAG pp. 142-155.

10. Day R., *A History of Software Maintenance for a Complex U.S. Army Battlefield Automated System*, IEEE Proceedings of Conference on Software Maintenance. IEEE, November 1985. pp. 181-187.

11. DeMarco T., *Controlling Software Projects: Management, Measurement & Estimation*, Yourdon Press, Prentice-Hall, Inc. 1982.

12. Dossey J.A. and others, *Discrete Mathematics*, Scott, Foresman and Company, 1987.

13. Douglas B.S., *Conceptual Level Design of a Design Database for the Computer-Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1989.

14. Heimbigner D. and Krane S., *A Graph Transform Model for Configuration Management Environments*, Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. 1988. pp. 216-225.

15. Horne C.G. and Seeger R., *An Advanced Configuration Management Tool Set*, IEEE Conference on Software Maintenance. IEEE, 1988. pp. 229 - 234.

16. *IEEE Guide to Software Configuration Management, ANSI/IEEE Std. 1042-1987*, Technical Committee on Software Engineering of the Computer Society of IEEE, 1988.

17. Lehman, M.M., and Belady L.A., *Program Evolution, Process of Software Change*, ACADEMIC PRESS 1985.

18. Lientz B.P., and Swanson E.B., *Software Maintenance Management*, ADDISON-WESLEY 1980.

19. Lientz B.P., and Swanson E.B., *Problems in Application Software Maintenance*, Communications of the ACM, Vol. 24, no 11, November 1981 pp. 763-769.

20. Lientz B.P., *Issues in Software Maintenance*, ACM Computing Surveys, Vol.15, No 3, Sep. 1983. pp. 171-178.

21. Luqi, *Computer Aided Maintenance of Prototype Systems*, Technical Report NPS-52-88-037, Naval Postgraduate School, Monterey, CA. September 1988

22. Luqi, *Software Evolution via Prototyping*, Technical Report NPS-52-88-039, Naval Postgraduate School, Monterey, CA. September 1988

23. Martin R.J., and McClure C.L., *Software Maintenance*, Prentice-Hall, Inc. 1983.

24. Martin R.J., and Osborne W.M., *Guidance on Software Maintenance*, National Bureau of Standards, U.S. Departments of Commerce, December 1983.

25. McClure C.L., *Managing Software Development and Maintenance*, VAN NOSTRAND REINHOLD Co. 1981.

26. Parkih G. (ed.), *Techniques of Program and System Maintenance*, Winthrop Publishers, Inc. 1982.

27. Rand P.H., *Seven Ways to Cut SOftware Maintenance Costs*, Datamation, July 15 1987. pp. 128-131.

28. Richardson G.L., and Butler C.W., *Organization Issues of Effective Maintenance Management*, AFIPS, Proceedings of the National Computer Conference, Vol. 52. 1983. pp. 157-161.

29. Roberts F.S., *Discrete Mathematical Models*, Prentice-Hall, Inc 1976.

30. Schaefer H., *Metrics for Optimal Maintenance Management*, IEEE Proceedings of Conference on Software Maintenance. IEEE, November 1985. pp. 114-119.

31. Schneidwind N.F., *The State of Software Maintenance*, IEEE Transactions on Software Engineering, Vol. SE-13 No. 3. March 1987.

32. Swanson E.B., *The Dimensions of Maintenance*, Proceedings of 2nd International Conference on Software Engineering. IEEE. October 1976. pp. 492-497.

33. Taylor B., *A Database Approach to Configuration Management for Large Projects*, IEEE Proceedings of Conference on Software Maintenance.IEEE, November 1985. pp. 15-23.

34. Tichy W.F., *Design, Implementation and Evaluation of a Revision Control System*, Proceedings of 6-th Software Engineering Conference. IEEE. September 1982.

35. Tichy W.F., *RCS - A System for Version Control*, Software - Practice and Experience, Vol. 15(7). July 1985. pp. 637-654.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Center for Naval Analysis 4401 Ford Avenue Alexandria, Virginia 22302-0268	1
4.	Research Administration, Code 012 Naval Postgraduate School Monterey, California 93943	1
5.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, California 93943	1
6.	Commanding Officer System Maintenance Division Israeli Air Force via Israeli Air Attache Embassy of Israel 3514 International Drive, N.W. Washington, D.C. 20008	3
7.	Prof. Luqi, Code CS/Lq Computer Science Department Naval Postgraduate School Monterey, California 93943	3
8.	Chief of Naval Research 800 N. Quincy Street Arlington, Virginia 22217	1
9.	Carnegie Mellon University Software Engineering Institute Department of Computer Science Attn. Dr. E. Borison Pittsburgh, Pennsylvania 15260	1

10. Commanding Officer — 1
Naval Research Laboratory, Code 5150
Attn. Dr. Elizabeth Wald
Washington, D.C. 20375-5000

11. Defense Advanced Research Projects Agency (DARPA) — 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. B. Boehm
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

12. Attn: Mr. William McCoy — 1
Code K54, NSWC
Dahlgren, Virginia 22448

13. Defense Advanced Research Projects Agency (DARPA) — 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

14. Defense Advanced Research Projects Agency (DARPA) — 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

15. Defense Advanced Research Projects Agency (DARPA) — 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

16. Chief of Naval Operations — 1
Attn: Dr. R. M. Carroll (OP-01B2)
Washington, D.C. 20350

17. Dr. Robert M. Balzer — 1
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Ray, California 90292-6695

18. Editor-in-Chief, IEEE Software — 1
Attn. Dr. Ted Lewis
Oregon State University
Computer Science Department
Corvallis, Oregon 97331

19. IBM T. J. Watson Research Center — 1
Attn. Dr. A. Stoyenko
P.O. Box 704
Yorktown Heights, New York 10598

20. International Software Systems Inc. — 1
12710 Research Boulevard, Suite 301
Attn. Dr. R. T. Yeh
Austin, Texas 78759

21. Kestrel Institute — 1
Attn. Dr. C. Green
1801 Page Mill Road
Palo Alto, California 94304

22. MCC AI Laboratory — 1
Attn. Dr. Michael Gray
3500 West Balcones Center Drive
Austin, Texas 78759

23. National Science Foundation — 1
Division of Computer and Computation Research
Attn. Tom Keenan
Washington, D.C. 20550

24. Naval Ocean Systems Center — 1
Attn: Linwood Sutton, Code 423
San Diego, California 92152-5000

25. Naval Ocean Systems Center — 1
Attn. Les Anderson, Code 413
San Diego, California 92152-5000

26. Naval Sea Systems Command — 1
Attn: Capt A. Thompson
National Center #2, Suite 7N06
Washington, D.C. 22202

27. NAVSEA, PMS-4123H — 1
Attn. William Wilder
Arlington, Virginia 22202-5101

28. New Jersey Institute of Technology — 1
Computer Science Department
Attn. Dr. Peter Ng
Newark, New Jersey 07102

29. Office of Naval Research — 1
Computer Science Division, Code 1133
Attn. Dr. Van Tilborg
800 N. Quincy Street
Arlington, Virginia 22217-5000

30. Office of Naval Research — 1
Computer Science Division, Code 1133
Attn. Dr. R. Wachter
800 N. Quincy Street
Arlington, Virginia 22217-5000

31. Office of Naval Research — 1
Applied Mathematics and Computer Science
Attn. J. Smith, Code 1211
800 N. Quincy Street
Arlington, Virginia 22217-5000

32. Software Group, MCC — 1
9430 Research Boulevard
Attn. Dr. L. Belady
Austin, Texas 78759

33. University of California at Berkeley — 1
Department of Electrical Engineering and Compute Science
Computer Science Division
Attn. Dr. C.V. Ramamoorthy
Berkeley, California 90024

34. Attn: Dr. Mike Reiley — 1
Fleet Combat Directicnal Systems Support Activity
San Diego, California 92147-5081

35. Chief of Naval Operations — 1
Attn. Dr. Earl Chavis (OP-162)
Washington, D.C. 20350

36. Steve Huseth — 1
Honeywell Systems & Research Center
Mpls, Minnesota 55418

37. Attn: George Sumiall — 1
US Army Headquarters
CECOM
AMSEL-RD-SE-AST-SE
Fort Monmouth, New Jersey 07703-5000

38. Attn: Joel Trimble — 1
1211 South Fern Street, C107
Arlington, Virginia 22202

39. Attn: Dr. David Hislop — 1
United States Laboratory Command
Army Research Office
P. O. Box 12211
Research Triangle Park, North Carolina 27709-2211

40. Attn: Dr. Phil Hwang — 1
NSWC, U-33
Silver Spring, Maryland 20903-5000

41. Attn: Dr. Abraham Waksman — 1
Computer Science and Artificial Intelligence
Department of the Air Force
Bolling Air Force Base, D.C. 20332-6448

42. Professor Valdis Berzins, Code CS/Be — 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

43. Professor Kim Hefner, Code MA/HK — 1
Department of Mathematics
Naval Postgraduate School
Monterey, California 93943

44. Professor Tarek Abdel-Hamid, Code AS/Ah — 1
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93943

45. Dr. Amiram Yehudai — 1
Tel Aviv University
School of Mathematical Sciences
Department of Computer Science
Tel Aviv, Israel 69978

46. Prof. D. Beery — 1
Department of Computer Science
University of California
Los Angeles, California 90024

47. Professor Moshe Zviran, Code AS/Zv — 1
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93943

48. Capt J. A. Hernandez, Jr. — 1
IRMD/ASD/Code 743
Marine Corps Logistics Base
Albany, Georgia 31704

49. Major Isaak Mostov — 2
Israeli Air Force
via Israeli Air Attache
Embassy of Israel
3514 International Drive, N.W.
Washington, D.C. 20008